Proceedings of the 2019 miniKanren and Relational Programming Workshop

Proceedings of the 2019 miniKanren and Relational Programming Workshop

William E. Byrd
and
Nada Amin

TR-02-19

Computer Science Group
Harvard University
Cambridge, Massachusetts

Preface

This report aggregates the papers presented at the first miniKanren and Relational Programming Workshop, hosted on August 2nd, 2019 in Berlin, Germany and co-located with the twenty-second International Conference on Functional Programming.

The miniKanren and Relational Programming Workshop is a new workshop for the miniKanren family of relational (pure constraint logic programming) languages: miniKanren, microKanren, core.logic, OCanren, Guanxi, etc. The workshop solicits papers and talks on the design, implementation, and application of miniKanren-like languages. A major goal of the workshop is to bring together researchers, implementors, and users from the miniKanren community, and to share expertise and techniques for relational programming. Another goal for the workshop is to push the state of the art of relational programming—for example, by developing new techniques for writing interpreters, type inferencers, theorem provers, abstract interpreters, CAD tools, and other interesting programs as relations, which are capable of being "run backwards," performing synthesis, etc.

6 papers were submitted to the workshop, and each submission was reviewed by two to three members of the program committee. After deliberation, all submissions were accepted to the workshop.

In addition to the six full papers presented

- William E. Byrd gave a morning tutorial on miniKanren,
- Daniel P. Friedman and William E. Byrd gave a closing Q&A with audience.

Thanks to all presenters, participants, and members of the program committee.

William E. Byrd & Nada Amin

Program Committee

Claire Alvis, Sparkfund
Nada Amin, Harvard University (Program Chair)
Tom Gilray, University of Alabama at Birmingham
Jason Hemann, Northeastern University
Eric Holk, Google
Kanae Tsushima, National Institute of Informatics
William E. Byrd, University of Alabama at Birmingham (General Chair)

Contents

Towards a miniKanren with fair search strategies

KUANG-CHEN LU, Indiana University, USA
WEIXI MA, Indiana University, USA
DANIEL P. FRIEDMAN, Indiana University, USA

We describe fairness levels in disjunction and conjunction implementations. Specifically, a disjunction implementation can be fair, almost-fair, or unfair. And a conjunction implementation can be fair or unfair. We compare the fairness level of four search strategies: the standard miniKanren interleaving depth-first search, the balanced interleaving depth-first search, the fair depth-first search, and the standard breadth-first search. The two non-standard depth-first searches are new. And we present a new, more efficient and shorter implementation of the standard breadth-first search. Using quantitative evaluation, we argue that each depth-first search is a competitive alternative to the standard one, and that our improved breadth-first search implementation is more efficient than the current one.

1 INTRODUCTION

miniKanren is a family of relational programming languages. Friedman et al. [3, 4] introduce miniKanren and its implementation in *The Reasoned Schemer* and *The Reasoned Schemer, 2nd Ed* (TRS2). Hemann et al. [5] describe microKanren, a minimal core of miniKanren comprised of only 54 LOC, miniKanren has been implemented in many other languages, including multiple ones using the same language (e.g. OCanren[7]). As demonstrated in Byrd et al. [2], miniKanren can be used to naturally express difficult computations, such as using an interpreter to perform example-based program synthesis, or using a proof checker as a theorem prover. The papers, talks, and tutorials on miniKanren.org present many other unusual problems, and their solutions in miniKanren.

A subtlety arises when a $\texttt{cond}^e$ contains many clauses: not every clause has an equal chance of contributing to the result. As an example, consider the following relation $\texttt{repeat}^o$ and its invocation.

```
(defrel (repeat^o x out)
  (cond^e
    ((≡  (,x) out))
    ((fresh (res)
       (≡  (,x . ,res) out)
       (repeat^o x res)))))
> (run 4 q
    (repeat^o '* q))
'((*) (* *) (* * *) (* * * *))
```

Next, consider the following disjunction of invoking $\texttt{repeat}^o$ with four different letters.

Authors' addresses: Kuang-Chen Lu, Indiana University, USA, kl13@iu.edu; Weixi Ma, Indiana University, USA, mvc@iu.edu; Daniel P. Friedman, Indiana University, USA, dfried@indiana.edu.

miniKanren.org/workshop/2019/8-ART1

```
> (run 12 q
    (cond^e
      ((repeat^o 'a q))
      ((repeat^o 'b q))
      ((repeat^o 'c q))
      ((repeat^o 'd q))))
```

cond^e intuitively relates its clauses with logical `or`. And thus an unsuspicious beginner would expect each letter to contribute equally to the result, as follows.

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

The cond^e in TRS2, however, generates a less expected result.

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

The miniKanren in TRS2 implements **i**nterleaving DFS (DFS$_i$), the cause of this unexpected result. With this search strategy, each cond^e clause takes half of its received computational resources and passes the other half to its following clauses, except for the last clause that takes all resources it receives. In the example above, the `a` clause takes half of all resourses. And the `b` clause takes a quarter. Thus `c` and `d` barely contribute to the result.

DFS$_i$ is sometimes powerful for an expert. By carefully organizing the order of cond^e clauses, a miniKanren program can explore more "interesting" clauses than those uninteresting ones, and thus use computational resources efficiently.

DFS$_i$ is not always the best choice. For instance, it might be less desirable for novice miniKanren users—understanding implementation details and fiddling with clause order is not their first priority. There is another reason that miniKanren could use more search strategies than just DFS$_i$. In many applications, there does not exist one order that serves all purposes. For example, a relational dependent type checker contains clauses for constructors that build data and clauses for eliminators that use data. When the type checker is generating simple and shallow programs, the clauses for constructors had better be at the top of the cond^e expression. When performing proof searches for complicated programs, the clauses for eliminators had better be at the top of the cond^e expression. With DFS$_i$, these two uses cannot be efficient at the same time. In fact, to make one use efficient, the other one must be more sluggish. Boskin et al. [1] propose and implement a means to eliminate or re-order disjunctive clauses to accommodate varying search needs such as these.

The specification that gives every clause in the same cond^e equal "search priority" is fair `disj`. And search strategies with almost-fair `disj` give every clause similar priority. Fair `conj`, a related concept, is more subtle. We cover it in the next section.

Our research compares four search strategies with different features in fairness (Table 1). To summarize our contributions, we

- propose and implement **b**alanced **i**nterleaving depth-first search (DFS$_{bi}$)
- propose and implement **f**air depth-first search (DFS$_f$)
- implement in a new way the standard breath-first search. We refer to BFS$_{ser}$ as the original implementation by Seres et al. [9] and BFS$_{imp}$ as our new one. When we use BFS without subscripts, we mean both BFS$_{ser}$

and BFS$_{imp}$. We formally prove that the two implementations are semantically equivalent, however, BFS$_{imp}$ runs faster in all benchmarks and is shorter.

Source code of implementations, examples, benchmarks, and formal proofs are available at the following URL:

https://github.com/LuKC1024/Towards-a-miniKanren-with-fair-search-strategies

Search Strategies	disj	conj
DFS$_i$	unfair	unfair
DFS$_{bi}$	almost-fair	unfair
DFS$_f$	fair	unfair
BFS	fair	fair

Table 1. Fairness of all search strategies

2 SEARCH STRATEGIES AND FAIRNESS

In this section, we define fairness levels in disjunction and conjunction implementations. Specifically, a disjunction implementation can be fair, almost-fair, or unfair. And a conjunction implementation can be fair or unfair. Fairness, intuitively, measures how evenly a search strategy allocates computational resource to "sibling" spaces.

Before going further into fairness, we give a short review of the terms: *state*, *space*, and *goal*. A *state* is a collection of constraints. (Here, we restrict constraints to unification constraints.) Every answer corresponds to a state. A *space* is a collection of states. And a *goal* is a function from a state to a space. Every state in the output space includes the input state and possibly more constraints.

Now we elaborate fairness by running more queries about `repeat`^{o}. We never use `run`* here because fairness is more interesting when we ask for a bounded number of answers. It is perfectly fine, however, to use `run`* with any search strategy.

2.1 Fair `disj`

Given the following program, it is natural to expect lists of each letter to constitute 1/4 in the answer list. DFS$_i$, TRS2's search strategy, however, results in many more lists of `a`s than lists of other letters. And some letters (e.g. `c` and `d`) are rarely seen. The more clauses, the worse the situation.

```
;; DFSi (unfair disj)
> (run 12 q
    (cond^e
      ((repeat^o 'a q))
      ((repeat^o 'b q))
      ((repeat^o 'c q))
      ((repeat^o 'd q))))
'((a) (a a) (b) (a a a)
  (a a a a) (b b)
  (a a a a a) (c)
  (a a a a a a) (b b b)
  (a a a a a a a) (d))
```

Under the hood, the cond^e here allocates computational resources to four trivially different spaces. The unfair `disj` in DFS$_i$ allocates many more resources to the first space. On the contrary, fair `disj` would allocate resources evenly to each space.

```
;; DFS_f (fair disj)
> (run 12 q
    (cond^e
      ((repeat^o 'a q))
      ((repeat^o 'b q))
      ((repeat^o 'c q))
      ((repeat^o 'd q))))
'((a) (b) (c) (d)
  (a a) (b b) (c c) (d d)
  (a a a) (b b b) (c c c) (d d d))
```

```
;; BFS (fair disj)
> (run 12 q
    (cond^e
      ((repeat^o 'a q))
      ((repeat^o 'b q))
      ((repeat^o 'c q))
      ((repeat^o 'd q))))
'((a) (b) (c) (d)
  (a a) (b b) (c c) (d d)
  (a a a) (b b b) (c c c) (d d d))
```

Running the same program again with almost-fair `disj` (e.g. DFS$_{bi}$) gives a similar result, where `b` and `c` are swapped. Almost-fair, however, is not completely fair, as shown by the following example.

```
;; DFS_bi (almost-fair disj)
> (run 16 q
    (cond^e
      ((repeat^o 'a q))
      ((repeat^o 'b q))
      ((repeat^o 'c q))
      ((repeat^o 'd q))
      ((repeat^o 'e q))))
'((a) (c) (b)
  (a a) (c c) (b b) (d)
  (a a a) (c c c) (b b b) (e)
  (a a a a) (c c c c) (b b b b) (d d)
  (a a a a a))
```

DFS$_{bi}$ is fair only when the number of goals is a power of 2, otherwise, it allocates some goals with twice as many resources as the others. In the above example, where the cond^e has five clauses, DFS$_{bi}$ allocates more resources to the clauses of `a`, `b`, and `c`.

We end this subsection with precise definitions of all levels of `disj` fairness. Our definition of *fair* `disj` is slightly more general than the one in Seres et al. [9], which is only for binary disjunction. We generalize it to a multi-arity one.

Definition 2.1 (fair `disj`). *A* `disj` *is fair if and only if it allocates computational resources evenly to spaces produced by goals in the same disjunction (i.e., clauses in the same* cond^e*).*

Definition 2.2 (almost-fair `disj`). *A* `disj` *is almost-fair if and only if it allocates computational resources so evenly to spaces produced by goals in the same disjunction that the maximal ratio of resources is bounded by a constant.*

Definition 2.3 (unfair `disj`). *A* `disj` *is unfair if and only if it is not almost-fair.*

2.2 Fair conj

Given the following program, it is natural to expect lists of each letter to constitute 1/4 in the answer list. Search strategies with unfair conj: DFS_i, DFS_{bi}, and DFS_f, however, results in many more lists of a's than lists of other letters. And some letters are rarely seen. Here again, as the number of clauses grows, the situation worsens.

Although some strategies have a different level of fairness in disj, they have the same behavior when there is no call to a relational definition in cond^e clauses, including this case.

```
;; DFSi (unfair conj)
> (run 12 q
    (fresh (x)
      (conde
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeato x q)))
'((a) (a a) (b) (a a a)
  (a a a a) (b b)
  (a a a a a) (c)
  (a a a a a a) (b b b)
  (a a a a a a a) (d))
```

```
;; DFSf (unfair conj)
> (run 12 q
    (fresh (x)
      (conde
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeato x q)))
'((a) (a a) (b) (a a a)
  (a a a a) (b b)
  (a a a a a) (c)
  (a a a a a a) (b b b)
  (a a a a a a a) (d))
```

```
;; DFSbi (unfair conj)
> (run 12 q
    (fresh (x)
      (conde
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeato x q)))
'((a) (a a) (b) (a a a)
  (a a a a) (b b)
  (a a a a a) (c)
  (a a a a a a) (b b b)
  (a a a a a a a) (d))
```

Under the hood, the cond^e and the call to repeat^o are connected by conj. The cond^e goal outputs a space including four trivially different states. Applying the next conjunctive goal, (repeato x q), produces four trivially different spaces. In the examples above, all search strategies allocate more computational resources to the space of a. On the contrary, fair conj would allocate resources evenly to each space. For example,

```
;; BFS (fair conj)
> (run 12 q
    (fresh (x)
      (conde
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeato x q)))
'((a) (b) (c) (d)
  (a a) (b b) (c c) (d d)
  (a a a) (b b b) (c c c) (d d d))
```

A more interesting situation is when the first conjunct produces an unbounded number of states. Consider the following example: a naive specification of fair conj might require search strategies to produce all sorts of singleton lists, but there would not be any lists of length two or longer, which makes the strategies incomplete. A search strategy is *complete* if and only if "every correct answer would be discovered after some finite time" [9], otherwise, it is *incomplete*. In the context of miniKanren, a search strategy is complete means that every correct answer has a position in large enough answer lists.

```
;; naively fair conj
> (run 6 q
    (fresh (xs)
      (conde
        ((repeato 'a xs))
        ((repeato 'b xs)))
      (repeato xs q)))
'(((a)) ((b))
  ((a a)) ((b b))
  ((a a a)) ((b b b)))
```

Our solution requires a search strategy with *fair* `conj` to organize states in buckets in spaces, where each bucket is a finite collection of states and every space contains possibly infinite buckets, and to allocate resources evenly among spaces derived from states in the same bucket. It is up to a search strategy designer to decide by what criteria to put states in the same bucket, and how to allocate resources among spaces related to different buckets.

BFS puts states of the same cost in the same bucket, and allocates resources carefully among spaces related to different buckets such that it produces answers in increasing order of cost. The *cost* of an answer is its depth in the search tree (i.e., the number of calls to relational definitions required to find the answer) [9]. In the above examples, the cost of each answer is equal to their lengths because we need to apply `repeato` n times to find an answer of length n. In the following example, every answer is a list of a list of symbols, where inner lists in the same outer list are identical. Here the cost of each answer is equal to the length of its inner lists plus the length of its outer list. For example, the cost of `((a) (a))` is $1 + 2 = 3$.

```
;; BFS (fair conj)
> (run 12 q
    (fresh (xs)
      (conde
        ((repeato 'a xs))
        ((repeato 'b xs)))
      (repeato xs q)))
'(((a)) ((b))
  ((a) (a)) ((b) (b))
  ((a a)) ((b b))
  ((a) (a) (a)) ((b) (b) (b))
  ((a a) (a a)) ((b b) (b b))
  ((a a a)) ((b b b)))
```

We end this subsection with precise definitions of all levels of `conj` fairness.

Definition 2.4 (fair `conj`). *A* `conj` *is fair if and only if it allocates computational resources evenly to spaces produced from states in the same bucket. A bucket is a finite collection of states. And search strategies with fair* `conj` *should represent spaces with possibly unbounded collections of buckets.*

Definition 2.5 (unfair `conj`). *A* `conj` *is unfair if and only if it is not fair.*

```
#| Goal × Goal → Goal |#
(define (disj₂ g₁ g₂)
  (lambda (s)
    (append^∞ (g₁ s) (g₂ s))))

#| Space × Space → Space |#
(define (append^∞ s^∞ t^∞)
  (cond
    ((null? s^∞) t^∞)
    ((pair? s^∞)
     (cons (car s^∞)
       (append^∞ (cdr s^∞) t^∞)))
    (else (lambda ()
            (append^∞ t^∞ (s^∞))))))

(define-syntax disj
  (syntax-rules ()
    ((disj) (fail))
    ((disj g₀ g ...) (disj+ g₀ g ...))))

(define-syntax disj+
  (syntax-rules ()
    ((disj+ g) g)
    ((disj+ g₀ g₁ g ...) (disj₂ g₀ (disj+ g₁ g ...)))))
```

Fig. 1. implementation of DFS_i (Part I)

3 INTERLEAVING DEPTH-FIRST SEARCH

In this section, we review the implementation of DFS_i. We focus on parts that are relevant to other strategies. TRS2, chapter 10 and the appendix, "Connecting the wires", provide a comprehensive description of the miniKanren implementation but limited to unification constraints (≡). Fig. 1 and Fig. 2 show parts that are later compared with other search strategies. We follow some conventions to name variables: ss name states; gs (possibly with subscript) name goals; variables ending with $^\infty$ name spaces. Fig. 1 shows the implementation of `disj`. The first function, $\texttt{disj}_2$, implements binary disjunction. It applies the two disjunctive goals to the input state `s` and composes the two resulting spaces with $\texttt{append}^\infty$. The following syntax definitions say `disj` is right-associative. Fig. 2 shows the implementation of `conj`. The first function, $\texttt{conj}_2$, implements binary conjunction. It applies the *first* goal to the input state, then applies the second goal to states in the resulting space. The helper function $\texttt{append-map}^\infty$ applies its input goal to states in its input space and composes the resulting spaces. It reuses $\texttt{append}^\infty$ for space composition. The following syntax definitions say `conj` is also right-associative.

```
#| Goal × Goal → Goal |#
(define (conj₂ g₁ g₂)
  (lambda (s)
    (append-map∞ g₂ (g₁ s))))

#| Goal × Space → Space |#
(define (append-map∞ g s∞)
  (cond
    ((null? s∞) '())
    ((pair? s∞)
     (append∞ (g (car s∞))
       (append-map∞ g (cdr s∞))))
    (else (lambda ()
            (append-map∞ g (s∞))))))

(define-syntax conj
  (syntax-rules ()
    ((conj) (fail))
    ((conj g₀ g ...) (conj+ g₀ g ...))))

(define-syntax conj+
  (syntax-rules ()
    ((conj+ g) g)
    ((conj+ g₀ g₁ g ...) (conj₂ g₀ (conj+ g₁ g ...)))))
```

Fig. 2. implementation of DFS_i (Part II)

4 BALANCED INTERLEAVING DEPTH-FIRST SEARCH

DFS_{bi} has almost-fair `disj` and unfair `conj`. Its implementation differs from DFS_i's in the `disj` macro. When there are one or more disjunctive goals, the new `disj` builds a balanced binary tree whose leaves are the goals and whose nodes are disj_2s, hence the name of this search strategy. In contrast, the `disj` in DFS_i constructs the binary tree in a particularly unbalanced form. We list the new `disj` with its helper in Fig. 3. The new helper, `disj+`, takes two additional 'arguments'. They accumulate goals to be put in the left and right subtrees. The first clause handles the case where there is only one goal. In this case, the tree is the goal itself. When there are more goals, we partition the list of goals into two sublists of roughly equal lengths and recur on the two sublists. We move goals to the accumulators in the last clause. As we are moving two goals each time, there are two base cases: (1) no goal remains; (2) one goal remains. We handle these two new base cases in the second clause and the third clause, respectively.

5 FAIR DEPTH-FIRST SEARCH

DFS_f has fair `disj` and unfair `conj`. Its implementation differs from DFS_i's in disj_2 (Fig. 4). The new disj_2 calls a new and fair version of append^∞. $\text{append}^\infty_{fair}$ immediately calls its helper, `loop`, with the first argument, `s?`,

```
(define-syntax disj
  (syntax-rules ()
    ((disj) fail)
    ((disj g0 g ...) (disj+ () () g0 g ...))))

(define-syntax disj+
  (syntax-rules ()
    ((disj+ () () g) g)
    ((disj+ (gl ...) (gr ...))
     (disj2 (disj+ () () gl ...)
            (disj+ () () gr ...)))
    ((disj+ (gl ...) (gr ...) g0)
     (disj2 (disj+ () () gl ...)
            (disj+ () () g0 gr ...)))
    ((disj+ (gl ...) (gr ...) ga g ... gz)
     (disj+ (gl ... ga) (gz gr ...) g ...))))
```

Fig. 3. implementation of DFS_{bi}

```
#| Goal × Goal → Goal |#
(define (disj2 g1 g2)
  (lambda (s)
    (append^∞_fair (g1 s) (g2 s))))

#| Space × Space → Space |#
(define (append^∞_fair s^∞ t^∞)
  (let loop ((s? #t) (s^∞ s^∞) (t^∞ t^∞))
    (cond
      ((null? s^∞) t^∞)
      ((pair? s^∞)
       (cons (car s^∞)
         (loop s? (cdr s^∞) t^∞)))
      (s? (loop #f t^∞ s^∞))
      (else (lambda ()
              (loop #t (t^∞) (s^∞)))))))
```

Fig. 4. implementation of DFS_{f}

initialized to `#t`, which indicates that we haven't swapped s^∞ and t^∞. The swapping happens at the third `cond` clause in `loop`, where `s?` is updated accordingly. The first two `cond` clauses essentially copy the `car`s and stop recursion when one of the input spaces is obviously finite. The third clause, as we mentioned above, is just for

```
#| Goal × Space → Space |#
(define (append-map^∞_fair g s^∞)
  (cond
    ((null? s^∞) '())
    ((pair? s^∞)
     (append^∞_fair (g (car s^∞))
       (append-map^∞_fair g (cdr s^∞))))
    (else (lambda ()
            (append-map^∞_fair g (s^∞))))))
```

Fig. 5. stepping-stone toward BFS_{imp} (based on DFS_f)

swapping. When the fourth and last clause runs, we know that both $\texttt{s}^\infty$ and $\texttt{t}^\infty$ end with thunks, and that we have swapped them. In this case, we construct a new thunk. The new thunk swaps back the two spaces in the recursive call to `loop`. This is unnecessary for fairness—we do it to produce answers in a more readable order.

6 BREADTH-FIRST SEARCH

BFS has both fair `disj` and fair `conj`. Our first BFS implementation (Fig. 5) serves as a "stepping-stone" toward BFS_{imp}. It is so similar to DFS_f (not DFS_i) that we only need to apply two changes: (1) rename append-map^∞ to $\text{append-map}^\infty_{fair}$ and (2) replace append^∞ with $\text{append}^\infty_{fair}$ in $\text{append-map}^\infty_{fair}$'s body.

This implementation can be improved in two ways. First, as mentioned in subsection 2.2, BFS puts answers in buckets and answers of the same cost are in the same bucket. In the above implementation, however, it is not obvious how we manage cost information—the `car`s of a space have cost 0 (i.e., they are all in the same bucket), and every thunk indicates an increment in cost. It is even more subtle that $\text{append}^\infty_{fair}$ and the $\text{append-map}^\infty_{fair}$ respects the cost information. Second, $\text{append}^\infty_{fair}$ is extravagant in memory usage. It makes $O(n + m)$ new `cons` cells every time, where n and m are the sizes of the first buckets of two input spaces. DFS_f is also space extravagant.

In the following paragraphs, we first describe BFS_{imp} implementation that manages cost information in a more clear and concise way and is less extravagant in memory usage. Then we compare BFS_{imp} with BFS_{ser}.

We simplify the cost information by changing the `Space` type, modifying related function definitions, and introducing a few more functions. The new type of `Space` is a pair whose `car` is a list of answers (the bucket), and whose `cdr` is either `#f` or a thunk returning a space. The `#f` here means the space is obviously finite, just like empty list in other implementations. We list functions related to the pure subset in Fig. 6. The first three functions are space constructors. `none` makes an empty space; `unit` makes a space from one answer; and `step` makes a space from a thunk. The remaining functions are as before. Luckily, the change in $\text{append}^\infty_{fair}$ also fixes the miserable space extravagance—the use of `append` helps us to reuse the first bucket of $\texttt{t}^\infty$. Functions implementing impure features are in Fig. 7. The first function, `elim`, takes a space $\texttt{s}^\infty$ and two continuations `fk` and `sk`. When $\texttt{s}^\infty$ contains no answers, it calls `fk`. Otherwise, it calls `sk` with the first answer and the rest of the space. This function is similar to an eliminator of spaces, hence the name. The remaining functions are as before.

Kiselyov et al. [6] have demonstrated that a *MonadPlus* hides in implementations of logic programming systems. BFS_{imp} is not an exception: $\text{append-map}^\infty_{fair}$ is like `bind`, but takes arguments in reversed order; `none`, `unit`, and $\text{append}^\infty_{fair}$ correspond to `mzero`, `unit`, and `mplus`, respectively.

```
#| → Space |#
(define (none)
   (() . #f))

#| State → Space |#
(define (unit s)
   ((,s) . #f))

#| (→ Space) → Space |#
(define (step f)
   (() . ,f))

#| Space × Space → Space |#
(define (append^∞_fair s^∞ t^∞)
  (cons (append (car s^∞) (car t^∞))
    (let ((t1 (cdr s^∞)) (t2 (cdr t^∞)))
      (cond
        ((not t1) t2)
        ((not t2) t1)
        (else (lambda () (append^∞_fair (t1) (t2))))))))

#| Goal × Space → Space |#
(define (append-map^∞_fair g s^∞)
  (foldr
    (lambda (s t^∞)
      (append^∞_fair (g s) t^∞))
    (let ((f (cdr s^∞)))
      (step (and f (lambda () (append-map^∞_fair g (f))))))
    (car s^∞)))

#| Maybe Nat × Space → [State] |#
(define (take^∞ n s^∞)
  (let loop ((n n) (vs (car s^∞)))
    (cond
      ((and n (zero? n)) '())
      ((pair? vs)
       (cons (car vs)
         (loop (and n (sub1 n)) (cdr vs))))
      (else (let ((f (cdr s^∞)))
              (if f (take^∞ n (f)) '()))))))
```

Fig. 6. New and changed functions in BFS_{imp} that implements pure features

```
#| Space × (State × Space → Space) × (↦ Space) → Space |#
(define (elim s∞ fk sk)
  (let ((ss (car s∞)) (f (cdr s∞)))
    (cond
      ((pair? ss) (sk (car ss) (cons (cdr ss) f)))
      (f (step (lambda () (elim (f) fk sk))))
      (else (fk)))))

#| Goal × Goal × Goal → Goal |#
(define (ifte g1 g2 g3)
  (lambda (s)
    (elim (g1 s)
      (lambda () (g3 s))
      (lambda (s0 s∞)
        (append-map∞fair g2
          (append∞fair (unit s0) s∞))))))

#| Goal → Goal |#
(define (once g)
  (lambda (s)
    (elim (g s)
      (lambda () (none))
      (lambda (s0 s∞) (unit s0)))))
```

Fig. 7. New and changed functions in BFS_{imp} that implement impure features

Now we compare the pure subset of BFS_{imp} with BFS_{ser}. We focus on the pure subset because BFS_{ser} is designed for a pure relational programming system. We prove in Coq that these two search strategies are semantically equivalent, since the result of `(run n ? g)` is the same either way. (See the GitHub repository for the formal proofs.) To compare efficiency, we translate BFS_{ser}'s Haskell code into Racket. (See the GitHub repository for the translated code.) The translation is direct due to the similarity of the two relational programming systems. The translated code is longer than BFS_{imp}. And it runs slower in all benchmarks. Details about differences in efficiency are in section 7.

7 QUANTITATIVE EVALUATION

In this section, we compare the efficiency of the search strategies. A concise description is in Table 2. A hyphen means "running out of 500 MB memory". The first two benchmarks are from TRS2. `revers`^{o} is from Rozplokhas and Boulytchev [8]. The next two benchmarks about generating quines are based on a similar test case in Byrd et al. [2]. We modify the relational interpreters because we don't have disequality constraints (e.g. `absent`^{o}). The sibling benchmarks differ in the `cond`^{e} clause order of their relational interpreters. The last two benchmarks are about synthesizing expressions that evaluate to `'(I love you)`. They are also based on a similar test case in Byrd et al. [2]. Again, we modify the relational interpreters for the same reason. And the sibling benchmarks

benchmark	size	DFS_i	DFS_{bi}	DFS_f	BFS_{imp}	BFS_{ser}
very-recursive^o	100000	184	180	510	554	1328
	200000	409	249	984	1063	2477
	300000	520	549	2713	2344	5815
append^o	100	25	26	24	23	89
	200	196	202	179	183	172
	300	556	536	540	560	524
revers^o	10	5	5	5	25	48
	20	46	48	47	4363	5145
	30	434	419	436	106746	151759
quine-1	1	109	123	28	-	-
	2	289	308	71	-	-
	3	522	541	99	-	-
quine-2	1	23	23	12	-	-
	2	52	51	24	-	-
	3	80	75	34	-	-
'(I love you)-1	999	76	96	64	260	635
	1999	158	210	115	332	669
	2999	453	330	279	331	672
'(I love you)-2	999	733	326	63	276	639
	1999	1430	859	114	334	674
	2999	2496	1137	280	327	683

Table 2. The results of a quantitative evaluation: running times of benchmarks in milliseconds

differ in the cond^e clause order of their relational interpreters. The first one has elimination rules (i.e. application, car, and cdr) at the end, while the other has them at the beginning. We conjecture that DFS_i would perform badly in the second case because elimination rules complicate the problem when synthesizing (i.e., our evaluation supports our conjecture.)

In general, variants of DFS usually performs better than BFS. The reason might be that BFS tends to remember more states at the same time. Among the three variants of DFS, which all have unfair conj, DFS_f is most resistant to clause permutation in quines and '(I love you)s, followed by DFS_{bi} then DFS_i. Thus, we consider DFS_{bi} and DFS_f competitive alternatives to DFS_i. Among the two implementations of BFS, BFS_{imp} constantly performs as well or better.

8 RELATED RESEARCH

In this section, we describe related research. Yang [10] points out that a disjunct complex would be 'fair' if it were a full and balanced tree. Seres et al. [9] describe BFS. We present another implementation. Our implementation is semantically equivalent to theirs. But, ours is shorter and performs better in comparison with a straightforward translation of their Haskell code. Rozplokhas and Boulytchev [8] address the non-commutativity of conjunction, while our work about disj fairness addresses the non-commutativity of disjunction.

9 CONCLUSION

We analyze the definitions of fairness. Implementation of disj can be fair, almost-fair, or unfair, depending on how evenly it allocates computational resources to spaces related to disjunctive goals. Implementation of conj

can be fair or unfair, depending on how evenly it allocates computational resources to spaces related to states in the same bucket. Our definition of fair `conj`, unlike the one by Seres et al. [9], is orthogonal with completeness.

We devise two new search strategies (i.e., DFS_{bi} and DFS_{f}) and devise a new implementation of BFS, BFS_{imp}. These strategies have different features in fairness: DFS_{bi} has an almost-fair `disj` and unfair `conj`. DFS_{f} has fair `disj` and unfair `conj`. BFS has both fair `disj` and fair `conj`. No search strategy here combines unfair `disj` and fair `conj`. This is because we haven't seen a case where these kinds of search strategies would be interesting.

Our quantitative evaluation shows that DFS_{bi} and DFS_{f} are competitive alternatives to DFS_{i}, the current miniKanren search strategy, and that BFS_{imp} is more practical than BFS_{ser}.

We prove formally that BFS_{imp} is semantically equivalent to BFS_{ser}. But, BFS_{imp} is shorter and performs better in comparison with a straightforward translation of their Haskell code.

Although there are very few benchmarks, this is preliminary work where we are making a point that certain levels of fairness come without cost in some cases, and that each of the search strategies: DFS_{i}, DFS_{bi}, DFS_{f}, and BFS, can co-exist inside one's head. Constructing a miniKanren with all levels of fairness is future work.

ACKNOWLEDGMENTS

We thank the program committee for their insightful observations. We also thank our reviewers, both known and anonymous, for their corrections and suggestions.

REFERENCES

[1] Benjamin Strahan Boskin, Weixi Ma, David Thrane Christiansen, and Daniel P. Friedman. 2018. A Surprisingly Competetive Conditional Operator: for miniKanrenizing the Inference Rules of Pie *(Scheme '18)*. St Louis, MO, USA.

[2] 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).

[3] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer.* The MIT Press.

[4] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer, Second Edition.*

[5] Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2016. A small embedding of logic programming with a simple complete search. In *Proceedings of the 12th Symposium on Dynamic Languages - DLS 2016*. ACM Press. https://doi.org/10.1145/2989225.2989230

[6] Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers:(functional pearl). *ACM SIGPLAN Notices* 40, 9 (2005), 192–203.

[7] Dmitry Kosarev and Dmitry Boulytchev. 2018. Typed embedding of a relational language in OCaml. *arXiv preprint arXiv:1805.11006* (2018).

[8] Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. ACM, 18.

[9] Silvija Seres, J Michael Spivey, and C. A. R. Hoare. 1999. Algebra of Logic Programming.. In *ICLP*. 184–199.

[10] Edward Z. Yang. 2010. Adventures in Three Monads. *The Monad. Reader Issue 15* (2010), 11.

First-order miniKanren representation: Great for tooling and search

GREGORY ROSENBLATT, University of Alabama at Birmingham, USA
LISA ZHANG, University of Toronto Mississauga, Canada
WILLIAM E. BYRD, University of Alabama at Birmingham, USA
MATTHEW MIGHT, University of Alabama at Birmingham, USA

We present a first-order implementation of miniKanren that makes it easy to build a miniKanren debugger, and allows any other processes (including a human or a neural network) to guide the search. Typical miniKanren implementations use procedures to represent data structures like goals and streams. Instead, our implementation uses Racket structs, which are transparent, decomposable, manipulable, and not coupled with a particular search strategy. We obtain this first-order implementation by carefully applying defunctionalization rules to a higher-order implementation, deriving two compatible versions with the same search behavior and comparable performance. Decoupling the search in the first-order implementation makes it possible to analyze, transform, and optimize a miniKanren program, even while that program is running. We use a "human guided" search as a miniKanren debugger, and to demonstrate the breadth of supported search strategies. The flexibility in how we interpret goals and streams opens up possibilities for new tools, and we hope to inspire the community to build better miniKanren tooling.

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

Additional Key Words and Phrases: miniKanren, microKanren, logic programming, relational programming, Scheme, Racket, program synthesis

1 INTRODUCTION

While miniKanren has been applied successfully to at least seven different programming problems [1], understanding how miniKanren arrives at answers is challenging for all but the simplest examples. Well-hidden typos and inefficient ordering of conjuncts can be difficult to detect. Why is it that after more than a decade of work, miniKanren still lacks developer tools, like a basic debugger?

A seemingly different challenge is supporting different miniKanren search strategies. Different strategies may be better-suited to particular miniKanren programs, so flexibility in choosing search strategies can make programs run faster. Unfortunately, in Byrd et al. [1], changing the search strategy required significant modifications to a miniKanren implementation, as well as a tedious, ad hoc reimplementation of a relational interpreter to work around the default search behavior.

These two problems—building a debugger, and implementing various search strategies—can seem very different on the surface. However, they are actually symptoms of one design decision common to most miniKanren implementations: *a higher-order representation of programs coupled to a particular search strategy.*

Authors' addresses: Gregory Rosenblatt, University of Alabama at Birmingham, USA, gregr@uab.edu; Lisa Zhang, University of Toronto Mississauga, Canada, lczhang@cs.toronto.edu; William E. Byrd, University of Alabama at Birmingham, USA, webyrd@uab.edu; Matthew Might, University of Alabama at Birmingham, USA, might@uab.edu.

miniKanren.org/workshop/2019/8-ART2

Most miniKanren implementations represent an executing relational program using *goals* and *streams*. In a *higher-order* implementation, goals and streams are represented using procedures. Procedures encapsulate computational behavior, defining the search strategy for satisfying constraints. Unfortunately, procedures can't be modified or decomposed. Printing one yields:

```
> (fresh (x y)
    (conde ((== 10 x) (== 20 y))
           ((== 30 x) (== 40 y))))
#<procedure:...>
```

Our proposal is to use a *first-order* program representation, where goals and streams are data structures not coupled to any search strategy. These data structures are decomposable, inspectable and manipulable:

```
> (fresh (x y)
    (conde ((== 10 x) (== 20 y))
           ((== 30 x) (== 40 y))))
#s(disj
   #s(conj #s(== 10 #s(var x 1)) #s(== 20 #s(var y 2)))
   #s(conj #s(== 30 #s(var x 1)) #s(== 40 #s(var y 2))))
```

The rest of the paper is organized as follows: we begin in Section 2 with an example of how a miniKanren stepper could be useful. Section 3 describes the requirements of a miniKanren implementation to support such a stepper. In Section 4, we implement a first-order version of miniKanren alongside a higher-order version with the same search behavior. In Section 5, we explain the program transformations that make the stepper output easier to understand. Section 6 describes applications of the first-order representation, such as building the stepper, using a neural guided search strategy, and using dynamic search strategies, as well as providing ideas for other miniKanren tools that have been difficult to implement in the past. We hope to inspire you to build these tools.

2 APPENDOH THE DEFICIENT: A MINIKANREN EMERGENCY

We begin this paper with a *miniKanren emergency* that we want you—dear readers—to help us resolve. Nub Let, a hapless but earnest beginner miniKanren programmer, has accidentally replaced our beloved "appendo the Magnificent" relation with **appendoh the deficient**.

As might be expected of such a dubious relation, appendoh has been failing left and right. Even worse, Nub Let forgot to email the code for appendoh as part of the bug report. Fortunately, Nub Let included a transcript of an interactive stepper session, enabled by using a first-order representation of miniKanren code.

Will you help us save this paper from the embarrassment of including a buggy five-line program?

Recall that a correct implementation of appendo looks like this:

```
(define-relation (appendo l s ls)
  (conde
    ((== '() l) (== s ls))
    ((fresh (a d res)
       (== `(,a . ,d) l)
       (== `(,a . ,res) ls)
       (appendo d s res)))))
```

The output of interactively stepping through an appendoh query is shown in Figure 1. The execution of the query is broken down into *steps*. Each *step* of evaluation corresponds to expanding one branch of a logical disjunction.

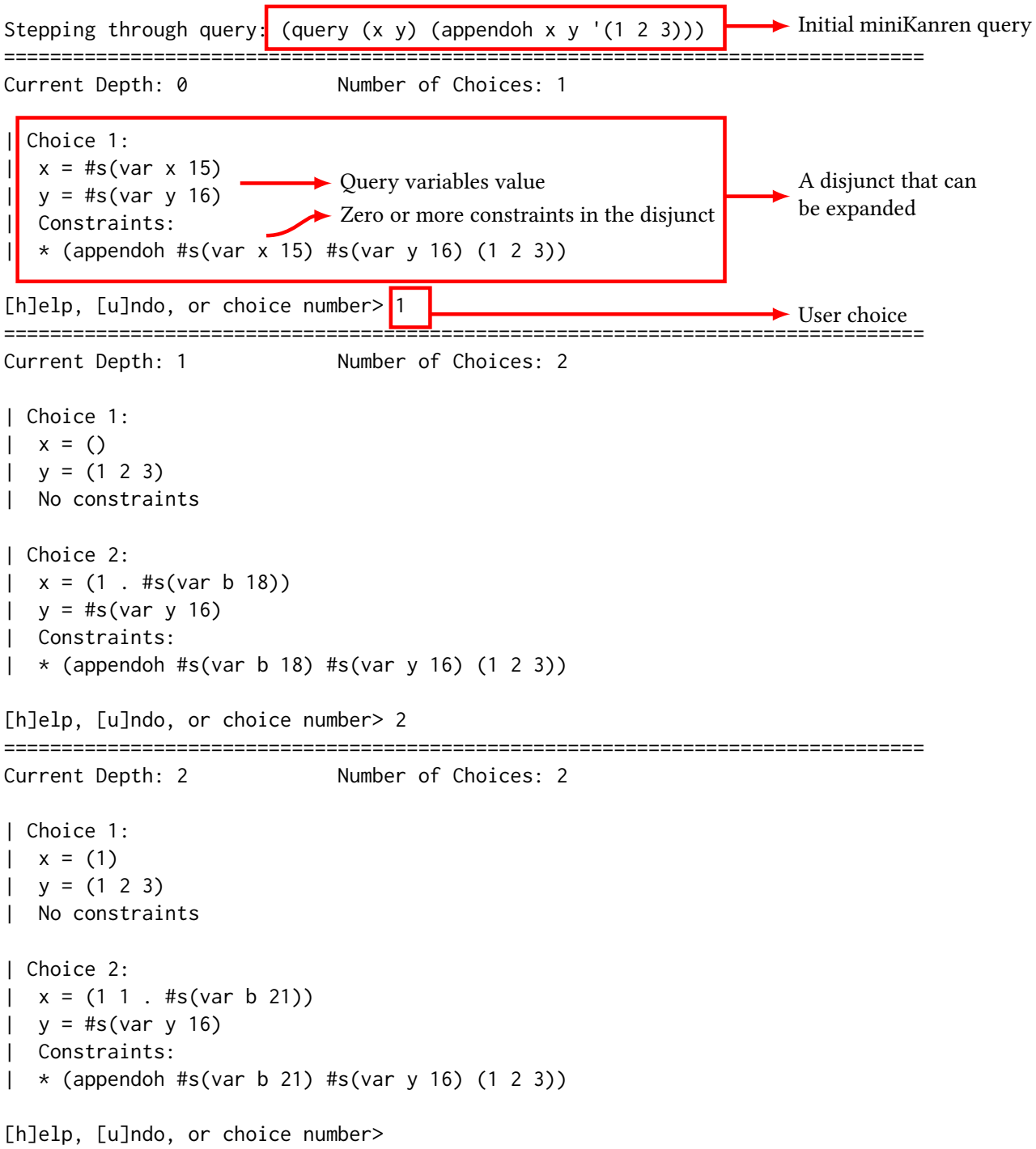

Fig. 1. Interactive stepper output for the buggy appendoh

Let us explain what you see in Figure 1: at each step, the user picks a disjunct to expand from a list of numbered *choices*. In the appendoh case, the disjuncts correspond to the conde branches in its definition, each of which constrains the query variables in some way. For each choice, the stepper shows the values the query variables will have if we take that choice. The stepper also shows zero or more constraints associated with a choice. These constraints correspond to goals that must be satisfied if we take the choice.

To prevent an explosion of choices, the stepper only displays the *new choices* that are descendants of the choice we just made in the previous step. We retain a history of user choices to provide the option of revisiting previous choices by backtracking. Not all choices will have descendants: some may fail entirely, and some successful choices may not have descendants.

Based on the stepper output in Figure 1 alone, can you deduce what the error is in the implementation of appendoh? The stepper output of the correctly specified appendo is in Appendix A.

It turns out that Nub Let hid the appendoh implementation in Appendix B. Even without the code, a keen reader may have noticed that in all the recursive calls to appendoh shown in the interactive stepper (look at the constraints), the value of the *last* argument never changes from `(1 2 3)`. This suggests that the bug is somewhere in the recursive call portion of the appendoh code, and that perhaps the last argument is incorrect.

3 DESIDERATA

The appendoh example is simple and has a linear search space. Even so, we hope that this example convinces you that a miniKanren stepper can be a useful debugging tool. In particular, being able to choose which disjuncts to explore is even more helpful for complex miniKanren programs with many disjuncts. Only some of the disjuncts might be buggy, so bugs would only manifest in queries that explore that disjunct. Being able to choose the order of execution allows the programmer to determine which portion of the search space is of interest.

To build a stepper like the one in Section 2, our miniKanren implementation must satisfy two requirements:

(1) **Streams should be decomposable, and thus inspectable.** We need some other data structure that, unlike procedures[1], can be programmatically transformed into a human-readable, textual representation.
(2) **Small-step execution should be possible.** We need to be able to take one small step to expand the chosen disjunct. Fine-grained control is important. The idea is not to search the disjunct exhaustively, but to see just a little further in order to inform our next decision. We may even end up deciding that continuing search of this disjunct will be a waste of effort, and prefer to revisit an earlier choice.

In short, we need the search state—the data representation of the stream—to be decoupled from the search behavior—the search strategy that miniKanren uses. Unfortunately, typical miniKanren implementations use a higher-order representation where these are hard to tease apart. Streams are represented as procedures that bundle the search state (the data) with the search behavior (a single interpretation of that data). To satisfy our requirements, we need a representation where the search state is accessible, so we can use it in at least two different ways: as information to be examined by the user, and as a stream to be stepped. If we instead use a first-order representation of the search state, we can view the stream as syntax, writing a different interpreter to satisfy each requirement.

If both functionalities are present, then the interactive stepper is straightforward to implement. The stepper extracts the list of disjuncts, presents these disjuncts as choices to the user, displays the query variable values and constraints associated with each choice, reads user input, takes a step to expand the chosen disjunct, and repeats until the user is satisfied.

[1]Technically, it is possible to use procedures implemented in an object-oriented style, with a method that prints a human-readable textual representation of its internal state. The issue with this approach is that supporting a new behavior requires implementing a new method. To avoid this issue, we could implement a method that provides a complete representation as a data structure suitable for any purpose. But at that point, we might as well just represent the stream as that data structure, and not use a procedure at all.

4 IMPLEMENTATIONS OF MINIKANREN

In this section, we build two miniKanren implementations, one of which uses a first-order representation of the search state to decouple it from the search behavior. That miniKanren implementation satisfies the two requirements from Section 3: that streams should be decomposable, and small-step executions should be possible.

Our plan is to build miniKanren on top of a version of μKanren. We start by implementing a higher-order μKanren to use as a familiar reference point. Our implementation is inspired by that of Hemann and Friedman [2], but modified for improved performance. We encourage interested readers to review Hemann and Friedman [2] for a detailed presentation of the μKanren language.

To obtain first-order μKanren, we will defunctionalize [4] the higher-order version. In particular, the first-order μKanren uses Racket's `struct`, which can be decomposed using `match` expressions (Section 4.1.1). This version of μKanren also has a small step interpreter `step` that performs a small, finite expansion of a stream (Section 4.3.1). For comparability, both μKanren versions implement the same search behavior.

The first-order and higher-order implementations share a portion of code, presented in Section 4.1. The remaining portions of both implementations are shown side by side in Figure 2 on page 8. We discuss the higher-order portion in Section 4.2, and compare with the first-order portion in Section 4.3. We recover a version of miniKanren in Section 4.4, compatible with both first-order and higher-order μKanren.

4.1 Common Implementation

The common portion defines *logic variables*, constraint *states*, *unification*, and *reification*:

4.1.1 Logic Variables. Logic variables[2] are defined using `struct`, which has the benefit of avoiding type collision with vectors. This allows support for vectors as terms, though for simplicity we do not include this support.

The declaration `(struct NAME (FIELD...) #:prefab)` defines a new record type, automatically generating a constructor named `NAME` and field accessors named `NAME-FIELD` for each `FIELD`. The `match` form can pattern match against values of this new type. Additionally, using `#:prefab` gives values of this type the printable notation `#s(NAME FIELD ...)`.

A logic variable contains the name it was given in the program, to help with debugging, and a non-negative integer index used to identify it. Unlike the implementation of Hemann and Friedman [2], a fresh logic variable is constructed with an index allocated from a shared, mutable counter, using `var/fresh`. This simplifies the role of states, which would otherwise carry a functional counter [2]. Additionally, without this simplification, a first-order implementation would be burdened with managing the functional counter to resolve unbound logic variables[3].

```
;; Logic variables
(struct var (name index) #:prefab)
(define (var=? x1 x2)
  (= (var-index x1) (var-index x2)))
(define initial-var (var #f 0))
(define var/fresh
  (let ((index 0))
    (lambda (name) (set! index (+ 1 index))
      (var name index))))
```

[2]Recall that a logic variable is a term that acts as a placeholder for an unknown value.

[3]Because we would like to implement many interpreters, this burden would multiply, because each interpreter would have to include support for resolving unbound logic variables.

4.1.2 States. States containing constraints are defined using `struct` and track equality information using a substitution. Other kinds of constraints could be supported, but for simplicity we will only consider equality constraints in this implementation.

Substitutions are implemented as in Hemann and Friedman [2], though we choose to enforce acyclic term structure using the `occurs?` check when extending a substitution via `extend-sub`. Our choice is consistent with typical miniKanren implementations.

```
;; States
(define empty-sub '())
(define (walk t sub)
  (let ((xt (and (var? t) (assf (lambda (x) (var=? t x)) sub))))
    (if xt (walk (cdr xt) sub) t)))
(define (occurs? x t sub)
  (cond ((pair? t) (or (occurs? x (walk (car t) sub) sub)
                       (occurs? x (walk (cdr t) sub) sub)))
        ((var? t)  (var=? x t))
        (else      #f)))
(define (extend-sub x t sub)
  (and (not (occurs? x t sub)) `((,x . ,t) . ,sub)))

(struct state (sub) #:prefab)
(define empty-state (state empty-sub))
```

4.1.3 Streams and Unification. Unification enforces equality between two terms and returns a substitution. It is defined as `unify`, and is implemented as in Hemann and Friedman [2].

```
;; Unification
(define (unify/sub u v sub)
  (let ((u (walk u sub)) (v (walk v sub)))
    (cond
      ((and (var? u) (var? v) (var=? u v)) sub)
      ((var? u)                            (extend-sub u v sub))
      ((var? v)                            (extend-sub v u sub))
      ((and (pair? u) (pair? v))           (let ((sub (unify/sub (car u) (car v) sub)))
                                             (and sub (unify/sub (cdr u) (cdr v) sub))))
      (else                                (and (eqv? u v) sub)))))
(define (unify u v st)
  (let ((sub (unify/sub u v (state-sub st))))
    (and sub (cons (state sub) #f))))
```

4.1.4 Reification. Reification normalizes the presentation of terms containing logic variables for easier reading and comparison. We implement `reify` as in Hemann and Friedman [2], except using a local, mutable counter to simplify allocation of normalized variable identifiers.

```
;; Reification
(define (walk* tm sub)
  (let ((tm (walk tm sub)))
    (if (pair? tm)
```

```
       `(,(walk* (car tm) sub) .  ,(walk* (cdr tm) sub))
       tm)))
(define (reified-index index)
  (string->symbol
    (string-append "_." (number->string index))))
(define (reify tm st)
  (define index -1)
  (walk* tm (let loop ((tm tm) (sub (state-sub st)))
              (define t (walk tm sub))
              (cond ((pair? t) (loop (cdr t) (loop (car t) sub)))
                    ((var? t)  (set! index (+ 1 index))
                               (extend-sub t (reified-index index) sub))
                    (else      sub)))))
(define (reify/initial-var st)
  (reify initial-var st))
```

4.2 Higher-order μKanren

Goals, immature streams, and stream maturation are implemented differently in higher- vs first-order μKanren. The remaining implementation of higher-order μKanren is shown on the left-hand-side of Figure 2 on page 8.

4.2.1 Goals. Goals include binary *disjunctions* and *conjunctions*, equality constraints, and calls to user-defined relations.

Binary disjunctions and conjunctions are constructed by `disj` and `conj`, which are defined as in Hemann and Friedman [2], except that our implementation introduces explicit pauses.

Equality constraints are constructed with `==`, which uses `unify`.

A call to a user-defined relation applies the relation to argument terms. Each call is represented as a *thunk*, wrapped with the constructor `relate`. A user-defined relation is expressed as a Racket procedure, such that forcing a call's thunk corresponds to applying the procedure to the call's arguments. (Higher-order μKanren places no restrictions on what a Racket procedure implementing a relation may do, but Section 4.3.2 will describe a few restrictions when defunctionalizing relation definitions in first-order μKanren.) A use of `relate` is also passed first-order metadata describing the call, but this information is discarded by our higher-order implementation.

4.2.2 Streams. Streams define a search strategy for finding answers, which are states that satisfy all of a program's constraints. For improved performance[4], we approximate the behavior of the interleaving search used in Byrd et al. [1].

A mature stream is either the empty stream `#f`, or a pair of an answer and a remaining stream. An immature stream is a thunk that has suspended the search before its completion. Forcing this thunk causes the search to continue. Streams will typically be constructed with the help of `mplus`, `bind`, and `pause`.

In our first-order implementation, we represent suspension more explicitly than in Hemann and Friedman [2]. A `pause` produces an immature stream by introducing a thunk that delays transition of a goal to a stream. When forced, the thunk applies the goal to a state.

The stream constructor `mplus` combines two streams. To improve answer diversity, our implementation of `mplus` interleaves more often to prevent highly productive branches from monopolizing search resources. We

[4]Compared with the search described by Hemann and Friedman [2], this implementation can synthesize quines about twice as fast, and twines and thrines nearly three times as fast, using a simple relational interpreter.

```
;; higher-order microKanren

(define (mature? s) (or (not s) (pair? s)))
(define (mature s)
  (if (mature? s) s (mature (s))))

(define (disj g1 g2)
  (lambda (st) (mplus (pause st g1)
                      (pause st g2))))
(define (conj g1 g2)
  (lambda (st) (bind (pause st g1) g2)))
(define (relate thunk _)
  (lambda (st) (pause st (thunk))))
(define (== t1 t2) (lambda (st) (unify t1 t2 st)))

(define (mplus s1 s2)
  (let ((s1 (if (mature? s1) s1 (s1))))
    (cond ((not s1) s2)
          ((pair? s1)
           (cons (car s1)
                 (lambda () (mplus s2 (cdr s1)))))
          (else (lambda () (mplus s2 s1))))))
(define (bind s g)
  (let ((s (if (mature? s) s (s))))
    (cond ((not s) #f)
          ((pair? s)
           (mplus (pause (car s) g)
                  (lambda () (bind (cdr s) g))))
          (else (lambda () (bind s g))))))
(define (pause st g) (lambda () (g st)))
```

```
1  ;; first-order microKanren
2  (struct disj   (g1 g2)                  #:prefab)
3  (struct conj   (g1 g2)                  #:prefab)
4  (struct relate (thunk description)      #:prefab)
5  (struct ==     (t1 t2)                  #:prefab)
6  (struct bind   (bind-s bind-g)          #:prefab)
7  (struct mplus  (mplus-s1 mplus-s2)      #:prefab)
8  (struct pause  (pause-state pause-goal) #:prefab)
9
10 (define (mature? s) (or (not s) (pair? s)))
11 (define (mature s)
12   (if (mature? s) s (mature (step s))))
13
14 (define (start st g)
15   (match g
16     ((disj g1 g2)
17      (step (mplus (pause st g1)
18                   (pause st g2))))
19     ((conj g1 g2)
20      (step (bind (pause st g1) g2)))
21     ((relate thunk _)
22      (pause st (thunk)))
23     ((== t1 t2) (unify t1 t2 st))))
24
25 (define (step s)
26   (match s
27     ((mplus s1 s2)
28      (let ((s1 (if (mature? s1) s1 (step s1))))
29        (cond ((not s1) s2)
30              ((pair? s1)
31               (cons (car s1)
32                     (mplus s2 (cdr s1))))
33              (else (mplus s2 s1)))))
34     ((bind s g)
35      (let ((s (if (mature? s) s (step s))))
36        (cond ((not s) #f)
37              ((pair? s)
38               (step (mplus (pause (car s) g)
39                            (bind (cdr s) g))))
40              (else (bind s g)))))
41     ((pause st g) (start st g))
42     (_            s)))
```

Fig. 2. Comparison of higher-order vs. first-order μKanren

interleave each time an answer is popped off of a mature stream, rather than popping all available answers before interleaving. To make this possible, `bind` needs to introduce a `pause` every time an answer is discovered.

Our implementation of `mplus` also delays forcing a child stream until it is ready to be examined. This small detail can significantly reduce wasted work and memory usage.

4.2.3 Stream maturation. Stream maturation is implemented by `mature`. In order to extract an answer from a stream, `mature` will continue forcing execution of an immature stream until it is mature.

4.3 First-order μKanren

We defunctionalize the higher-order representation of *goals* and *streams* to obtain the first-order implementation of μKanren shown in Figure 2. This new version of μKanren represents goal and streams as decomposable data structures, and interprets streams using `step`, with the help of a goal interpreter `start`.

The goal constructors `disj`, `conj`, `relate`, and `==`, along with the stream constructors `bind`, `mplus`, and `pause`, are now defined using `struct` to allow decomposition by pattern matching via the keyword `match`. Mature streams are represented the same way as in higher-order μKanren, as either the empty stream `#f`, or a pair of an answer and a remaining stream.

We add two new procedures, `step` and `start`, which are small-step interpreters for streams and goals respectively. These interpreters work together to implement the same interleaving search strategy defined by the higher-order representation. More specifically, `step` interprets an immature stream by making a finite amount of progress in searching for an answer. It interprets a mature stream by simply returning the stream. The interpreter `start` interprets a goal in the context of a state, producing a new stream.

4.3.1 Defunctionalizing goals and streams. Each higher-order goal and stream definition maps to a pattern matching clause in the corresponding interpreter. We have aligned the higher-order and first-order implementation listings in Figure 2 to make this correspondence easy to see: for each line in the higher-order definition on the left, scan along the same line to see the first-order version on the right.

We need to be careful when translating from higher-order to first-order. Failing to apply `step` at the right time may impede search progress, and change the order in which answers are found. Applying `step` too soon may lead to an infinite loop when a recursive relation is involved. These issues may not manifest immediately, so when they show up, it is unclear whether they are due to a particular query, the definition of a relation, or a problem with the search implementation. To avoid causing these issues when defunctionalizing the higher-order representation, we must match its order of evaluation. We follow these rewrite rules to defunctionalize:

Rule: Top-level higher-order constructors become `match` clauses

- Every top-level higher-order goal constructor becomes a `struct` pattern clause of the `match` expression in `start`.
- Every top-level higher-order stream constructor becomes a `struct` pattern clause of the `match` expression in `step`.
- In each case, the pattern decomposes the `struct`, naming its components the same way the higher-order constructors name their parameters. The clause bodies will then be defunctionalized.

Rule: Delaying goal/stream interpretation

- Anywhere that `lambda` is used in the higher-order implementation to delay construction of a goal or stream, we simply apply the corresponding `struct` constructor.
- **Justification**: though the higher-order constructors are procedures which immediately execute their interpretation when applied, the enclosing `lambda` leaves them inert until forced. Likewise, newly-constructed `struct` data remains inert until interpreted by `step` or `start`.

Rule: Interpreting a goal/stream

- Anywhere that the higher-order implementation does not delay construction of a goal, we apply `start` to the result of the corresponding `struct` constructor.
- Except in the case of `pause`, anywhere that the higher-order implementation does not delay construction of a stream, we apply `step` to the result of the corresponding `struct` constructor.
- Anywhere the higher-order implementation uses `pause`, we simply apply the `pause` `struct` constructor.
- **Justification**: We apply `start` and `step` in this way because the higher-order constructors also immediately execute their interpretation. But `pause` is an exception because it does not immediately execute its interpretation. Instead, it delays interpretation by producing an immature stream (a thunk).

Rule: Forcing immature streams

- Anywhere that the higher-order implementation forces progress of an immature stream, which corresponds to applying a thunk, we apply `step` to that stream to also force progress. Look at the definition of `mature` in Figure 2 for a simple example of this.

4.3.2 Defunctionalizing user-defined relations. As in the higher-order implementation, we represent calls to user-defined relations using thunks wrapped with the constructor `relate`. Forcing such a thunk applies a Racket procedure to the call's arguments. Unlike the higher-order implementation, uses of first-order `relate` will retain the call description metadata passed alongside the thunk. While an entirely first-order representation of user-defined relations is possible, we prefer retaining this use of thunks because it allows us to leverage Racket's implementation[5] of lexical scope and procedure abstraction, rather than reimplement this ourselves.

Despite continuing to use thunks to represent calls to relations, it is still possible to perform arbitrary manipulation of relation definitions if we make some assumptions about what such definitions may do. Specifically, we will assume the use of Racket is limited to constructing μKanren goals. When these goals include terms, the terms may be expressed using constants, references to lexical variables, pair construction, or fresh logic variable construction.

Given our assumptions, we can obtain a first-order representation of a user-defined relation by applying it to fresh logic variables to represent the parameters. Any goals in the result are those specified in the body of the relation. Any terms in those goals are also either those specified, or they are logic variables. If a logic variable is one of those used to represent a parameter of the relation, then the relation definition must reference that parameter at that location. All other logic variables were created fresh. Additionally, since each use of `relate` wraps metadata describing the call, we can always reconstruct a complete first-order description of a relation, even if it is recursive.

4.4 Recovering miniKanren

In Appendix C we implement miniKanren on top of μKanren, using `syntax-rules` definitions similar to those in Hemann and Friedman [2], with a notable addition: we define `query`, which describes the initial stream built by a corresponding `run`. Since we are interested in manipulating streams in more ways than just extracting answers, we need to separate stream construction from answer extraction. To extract answers from a stream, we provide `stream-take` such that:

```
(run N body ...) ==> (stream-take N (query body ...))
```

We will make use of the metadata wrapped by `relate` when visualizing program representation in Section 5. The metadata is a list containing the relation procedure (to allow unambiguous identification), the name of that procedure, and all of the argument terms.

[5]Hemann and Friedman [2] leverages Scheme in the same way.

4.5 Performance

When comparing the higher-order and first-order implementations, we notice a small performance difference when generating twines and thrines using a simple relational interpreter, and when running relational arithmetic benchmarks. On these tests, the first-order implementation is roughly 5 percent slower and spends about 10 percent more time in the garbage collector. Chez Scheme runs a Scheme version of our benchmarks about 25 percent faster than Racket, but the first-order vs. higher-order performance ratio remains about the same.

5 PROGRAM TRANSFORMATIONS

With the first-order miniKanren implementation given in Section 4, streams are decomposable (and therefore printable), and small-step executions are possible.

You might wonder whether we could stop here. After all, we can apply the procedure `step` to progress through a query, and print the structure of the stream at each step. Appendix D shows the output of stepping through the query `(query (xs) (appendo xs xs '(a a)))` using interleaving search.

Unfortunately, this presentation is intolerably verbose, even for such a simple query. It is difficult to extract useful information about actual available choices. In addition, many useless steps are taken to expand goals that are obviously failing. In order to obtain a simpler program representation that is actually human-readable, we need to **extract the useful information about each disjunct**, and **remove obviously failing disjuncts**.

This section recovers the useful trace information that we saw in Figure 1 by introducing stream transformations. Section 5.1 describes how to prune obviously failing disjuncts, so they need not be considered by a human. Section 5.2 describes how to transform the stream into disjunctive normal form (DNF), to provide a flat tree of disjuncts to choose from.

5.1 Pruning obvious failures

Some streams and goals in Appendix D are obviously failing. Examples include streams of the form `(bind #f _)`, `(mplus #f #f)`, goals of the form `(== A B)`, where `A` and `B` are unequal terms, and other streams and goals that depend on failing components. We would prefer to eliminate these immediately so we can focus on meaningful parts of the search state.

We can define a *pruning* transformation that uses equality information to identify constraints that will definitely fail. Since streams and goals depending on these constraints may fail to produce any results, we will prune them accordingly, producing a simplified stream.

Before taking a look at the implementation, it is important to consider that a transformation may change the observed behavior of our program. We should decide what sorts of changes we will tolerate. Viewing a miniKanren program as a search for answers that satisfy a query, we feel the following concessions are acceptable:

(1) Answers may be reordered.
(2) The contents of a state may be rearranged as long as its constraints express the same meaning.
(3) The number of steps required to find an answer may change, but will remain finite.
(4) Unproductive, infinite streams may become finite.

With that in mind, let us take a look at the implementation. The interpreters `prune/stream` and `prune/goal` partially evaluate == constraints, using the resulting `state` to propagate equality information. Any failing stream or goal will cause its parent `bind` and `conj` to also fail. A `mplus` or `disj` containing a failing child will be replaced by the remaining child.

```
(define (prune/stream s)
  (match s
    ((mplus s1 s2) (match (prune/stream s1)
```

```
                    (#f (prune/stream s2))
                    (s1 (match (prune/stream s2)
                          (#f s1)
                          (s2 (mplus s1 s2))))))
    ((bind s g)     (match (prune/stream s)
                    (#f          #f)
                    (`(,st . #f) (prune/goal st g))
                    ((pause st g1)
                     (match (prune/goal st g)
                       (#f           #f)
                       ((pause st g) (pause st (conj g1 g)))))
                    (s (match (prune/goal empty-state g)
                         (#f          #f)
                         ((pause _ _) (bind s g))))))
    ((pause st g)  (prune/goal st g))
    (`(,st . ,s)   `(,st . ,(prune/stream s)))
    (s             s)))

(define (prune/goal st g)
  (define (prune/term t) (walk* t (state-sub st)))
  (match g
    ((disj g1 g2)
     (match (prune/goal st g1)
       (#f (prune/goal st g2))
       ((pause st1 g1)
        (match (prune/goal st g2)
          (#f           (pause st1 g1))
          ((pause _ g2) (pause st (disj g1 g2)))))))
    ((conj g1 g2)
     (match (prune/goal st g1)
       (#f            #f)
       ((pause st g1) (match (prune/goal st g2)
                        (#f            #f)
                        ((pause st g2) (pause st (conj g1 g2)))))))
    ((relate thunk d) (pause st (relate thunk (prune/term d))))
    ((== t1 t2)
     (let ((t1 (prune/term t1)) (t2 (prune/term t2)))
       (match (unify t1 t2 st)
         (#f          #f)
         (`(,st . #f) (pause st (== t1 t2)))))))))
```

Since pruning performs obvious clean up work without spending steps to do so, pruning often leads to a trace containing fewer steps.

5.2 Disjunctive Normal Form

In Figure 1, we actually do one more thing—we provide the user with a flat list of choices. We can obtain those choices by first transforming the stream into disjunctive normal form (DNF).

The general DNF transformation rewrites goals and streams in the following way:

```
(conj (disj A B) C)             ==> (disj (conj A C)
                                          (conj B C))

(conj C (disj A B))             ==> (disj (conj C A)
                                          (conj C B))

(pause st (disj A B))           ==> (mplus (pause st A)
                                           (pause st B)

(bind (mplus A B) C)            ==> (mplus (bind A C)
                                           (bind B C))

(bind C (disj A B))             ==> (mplus (bind C A)
                                           (bind C B))
```

This transformation has the effect of lifting all disjunctions out of conjunctions.

```
(define (dnf/stream s)
  (define (push-pause st g)
    (match g
      ((disj g1 g2) (mplus (push-pause st g1) (push-pause st g2)))
      (g            (pause st g))))
  (match s
    ((bind s g)
     (let loop1 ((s (dnf/stream s)) (g (dnf/goal g)))
       (define (loop2 s g)
         (match g
           ((disj ga gb) (mplus (loop2 s ga) (loop2 s gb)))
           (g            (bind s g))))
       (match s
         ((mplus sa sb) (mplus (loop1 sa g) (loop1 sb g)))
         (`(,st . ,s)   (mplus (push-pause st g) (loop1 s g)))
         (s             (loop2 s g)))))
    ((pause st g)  (push-pause st (dnf/goal g)))
    ((mplus s1 s2) (mplus (dnf/stream s1) (dnf/stream s2)))
    (`(,st . ,s)   `(,st . ,(dnf/stream s)))
    (s             s)))

(define (dnf/goal g)
  (match g
    ((conj g1 g2)
     (let loop1 ((g1 (dnf/goal g1)) (g2 (dnf/goal g2)))
       (define (loop2 g1 g2)
```

```
      (match g2
        ((disj g2a g2b) (disj (loop2 g1 g2a) (loop2 g1 g2b)))
        (g2             (conj g1 g2))))
    (match g1
      ((disj g1a g1b) (disj (loop1 g1a g2) (loop1 g1b g2)))
      (g1             (loop2 g1 g2)))))
  ((disj g1 g2) (disj (dnf/goal g1) (dnf/goal g2)))
  (g           g)))
```

Aside from preparing a stream for choice extraction, DNF has the added benefit of allowing equality information to propagate more deeply, making pruning more effective. Consider the following goal:

```
(conj (disj (== x 5) A) B)
```

In this situation, information flow is impeded. Though we constrain `x` to be equal to 5, we cannot propagate this information into the goal `B`, because it is possible that goal `A` may not express the same constraint.

We can allow this information to flow more freely by lifting the disjunction out of the conjunction like so:

```
(disj (conj (== x 5) B)
      (conj A        B))
```

Because each child of the disjunction now has its own copy of the goal `B`, it is safe to propagate the equality information into the affected copy of `B`.

Another interesting property of DNF is that if we apply the full pruning transformation on a DNF stream, all remaining `==` constraints will be trivial in the sense that they are guaranteed to succeed, and will not provide new information. This is because all available equality information will have already been gathered in a `pause` state. These trivial constraints may be safely removed.

One downside of DNF transformation is that program representations can become larger due to copying children of conjunctions. In the worst case, the increase is exponential in the size of the original program. This potential increase in size may be mitigated by an increased likelihood of prunable, obvious failures, thanks to the improved flow of equality information.

Once we have a pruned DNF stream, we are finally in a position to provide choices of the form shown in Figure 1. In section 6.1 we will explain how we extract this list of choices from such a stream.

6 APPLICATIONS

6.1 The miniKanren stepper

In this section, we implement `explore/stream`, the miniKanren stepper described in Figure 1. The implementation of `explore/stream` is shown below. The main stepping loop follows the outline given at the end of Section 3. We describe each facet of this loop in chronological order.

The first thing the `explore/stream` loop does is use the procedure `stream->choices` to extract a list of choices from a stream. To prepare for extraction, we transform the stream into disjunctive normal form (DNF) using `dnf/stream`, and perform pruning with `prune/stream`. Recall that these procedures were implemented in Section 5. With a pruned DNF stream, we are guaranteed to find all disjunctions at the top (after any available answers), as a `mplus` tree, with leaves of the form `(pause STATE GOAL)`. Each of these leaves represents a choice, and all choices can be extracted while traversing the tree.

Once we have our list of choices, we need to present each choice in a digestible form. We handle this with `print-choice`. The `GOAL` in each choice is either a trivial `==` constraint (i.e., both terms are guaranteed to be equal), a `relate` call, or a `conj` tree whose leaves are goals of this form. Traversing this `conj` tree, as in `goal->constraints`, extracts these goals as a list of the choice's constraints. As mentioned in Section 5.2, pruning

in DNF has the emergent property that any remaining == constraints are guaranteed to be trivial, so we may discard them, allowing us to focus on the `relate` calls. Finally, pruning in DNF also guarantees that, of the variables mentioned in STATE, only those reachable from query variables are referenced in non-trivial constraints. Therefore, we can discard STATE, keeping only the query variable values.

With the interesting part of the implementation out of the way, all we have left to do is `read` user input and act accordingly. When the user makes a choice, we use `step` to expand the choice, providing the resulting stream as input to the next iteration of `explore/stream`.

```
(define (explore/stream qvars s)
  (define margin "| ")
  (define (pp prefix v) (pprint/margin margin prefix v))
  (define (pp/qvars vs)
    (define (qv-prefix qv) (string-append " " (symbol->string qv) " = "))
    (define qv-prefixes (and qvars (map qv-prefix qvars)))
    (if qv-prefixes
      (for-each (lambda (prefix v) (pp prefix v)) qv-prefixes vs)
      (for-each (lambda (v) (pp " " v)) vs)))
  (define (print-choice s)
    (match s
      ((pause st g)
       (pp/qvars (walked-term initial-var st))
       (define cxs (walked-term (goal->constraints st g) st))
       (unless (null? cxs)
         (displayln (string-append margin " Constraints:"))
         (for-each (lambda (v) (pp " * " v)) cxs))
       (when (null? cxs)
         (displayln (string-append margin " No constraints"))))))
  (let loop ((s (stream->choices s)) (undo '()))
    (define previous-choice
      (and (pair? undo)
           (let* ((i.s (car undo)) (i (car i.s)) (s (cdr i.s)))
             (list-ref (dropf s state?) (- i 1)))))
    (define results (takef s state?))
    (define choices (dropf s state?))
    (display "\n==================================")
    (displayln "==================================")
    (unless (= (length results) 0)
      (printf "Number of results: ~a\n" (length results))
      (for-each (lambda (st)
                  (pp/qvars (walked-term initial-var st))
                  (newline))
                results))
    (when (and previous-choice (null? results))
      (printf "Previous Choice:\n")
      (print-choice previous-choice)
      (newline))
```

```
(printf "Current Depth: ~a\n" (length undo))
(if (= 0 (length choices))
  (if (= (length results) 0)
    (printf "Choice FAILED!  Undo to continue.\n")
    (printf "No more choices available.  Undo to continue.\n"))
  (printf "Number of Choices: ~a\n" (length choices)))
(for-each (lambda (i s)
            (printf (string-append "\n" margin "Choice ~s:\n") (+ i 1))
            (print-choice s))
          (range (length choices)) choices)
(printf "\n[h]elp, [u]ndo, or choice number> ")
(define (invalid)
  (displayln "\nInvalid command or choice number.\nHit enter to continue.")
  (read-line) (read-line)
  (loop s undo))
(define i (read))
(cond ((eof-object? i) (newline))
      ((or (eq? i 'h) (eq? i 'help))
       (displayln
         (string-append "\nType either the letter 'u' or the"
                        " number following one of the listed choices."
                        "\nHit enter to continue."))
       (read-line) (read-line)
       (loop s undo))
      ((and (or (eq? i 'u) (eq? i 'undo)) (pair? undo))
       (loop (cdar undo) (cdr undo)))
      ((and (integer? i) (<= 1 i) (<= i (length choices)))
       (loop (stream->choices (step (list-ref choices (- i 1))))
             (cons (cons i s) undo)))
      (else (invalid))))
```

6.2 Search Strategies

A first-order miniKanren with the small-step interpreter `step` gives us fine-grained control over the search strategy. Even though we begin Section 4.2 by reimplementing the biased-interleaving search, we can use whatever search strategy we like. In this section, we describe a few different search strategies to illustrate the potential.

6.2.1 Human-guided search. One way of interpreting what we did in Section 6.1 is that we *hijacked the search strategy*. That is, the miniKanren stepper is actually an implementation of a human-guided search strategy! Since human behavior is as unpredictable as any program can be, the miniKanren stepper shows that any kind of search strategy is possible.

6.2.2 Neural-guided search. Another idea is to replace the human-guided search strategy with an artificial human, namely a neural network. In Zhang et al. [6] a neural network learns to guide the search of first-order miniKanren performing a Programming by Example (PBE) task. In such a task, miniKanren is used to synthesize a procedure specified using input/output pairs.

For example, if we have a relational interpreter (`eval-expo program environment output`), then an example PBE query with two input-output pairs is:

```
(query (p)
  (eval-expo p '(a) '(a a a))
  (eval-expo p '(b) '(b b b)))
```

The neural network evaluates each choice (disjunct) independently. It takes as input all the constraints corresponding to each choice, and chooses the highest-scoring choice to expand in each step.

There are several deviations from Section 6.1 required for this setup. First, unlike in Figure 1, the neural network can choose to expand any choice in each step, not just descendants of the previous step. So, the neural network would not need the "undo" functionality at all. This revised setup means that the neural network does not need to remember any past information, and can treat each step independently.

Another deviation is including a powerful strategy appropriate to PBE problems, which specializes the expansion behavior of `eval-expo` constraints. In a PBE problem, each input-output example imposes constraints on the same program to be synthesized, using `eval-expo`. However, a typical miniKanren search will attempt to satisfy each `eval-expo` constraint completely before moving on to the next. This is unfortunate because only information from one input-output example is considered at any given time, leading to deep exploration of impossible synthesis choices that could have been ruled out immediately. Instead, we implement an expansion step that will simultaneously expand all `eval-expo` constraints that share the same program argument. Combined with DNF transformation and pruning, simultaneous expansion leads to immediate feedback being provided by all input-output examples for each synthesis choice made. Aside from refuting bad choices more quickly, this simultaneous expansion allows the neural network to observe related constraints all at once.

The network in Zhang et al. [6] is trained on generated PBE problems to which we know the answer, so that the correct choice is known for each step. During training, the neural network guides the search by making choices, and incorrect choices are penalized. We refer interested readers to Zhang et al. [6] for more details about the model architecture and training.

6.2.3 Dynamic Goal Reordering. Though we have mostly discussed transformations while developing the interactive stepper, there are other ways to improve a search strategy. One source of possibilities is Byrd et al. [1], which makes use of a relational interpreter that performs various forms of conjunction reordering based on heuristics specific to the interpreter definition. For instance, goals are reordered based on determinism annotations on `conde` expressions, where at most one clause of a `conde` may succeed if we know the value of a particular subset of logic variables. The `conde` expressions recognized as deterministic are expanded before any others. This reordering would be more natural to implement with a decoupled representation, where it is easier to compose with transformations and other strategies.

6.3 More possibilities

Beyond improvements to the search, a first-order representation is a natural medium for program analysis and compilation. For instance, though we have not explored these ideas, we believe it should be possible to implement garbage collection of constraints and substitutions, and just-in-time compilation, possibly benefiting from mode analysis.

7 RELATED WORK

There have been previous unpublished implementations of miniKanren search using a first-order implementation. For example, Michael Ballantyne has implemented a small-step miniKanren interpreter that uses a first-order

representation.[6] Ballantyne has also implemented a big-step interpreter with a first-order representation of the search tree, in order to implement backjumping search [7]. Both of these implementations decouple the program representation from the search.

To our knowledge, this paper is the first first-order representation of a miniKanren search that preserves the exact search order of the standard higher-order search representation, or that provides rules for a defunctionalization that preserves this search order.

We are happy to report that in this same workshop, two other papers also make use of first-order representations. We hope this is the start of a new trend. Lozov et al. [3] performs supercompilation (as conjunctive partial deduction) using a first-order representation of goals that is similar to our representation of the search state. Rozplokhas et al. [5] also uses a similar first-order goal representation to formalize semantics for miniKanren.

8 CONCLUSION

Both higher-order and first-order representations of miniKanren have their advantages. For both pedagogical reasons and for the beauty and abstractness of the implementation, higher-order representations have been dominant in the miniKanren literature. On multiple occasions, however, implementors have had to switch to first-order representations in order to inspect or control the execution of miniKanren code.

Currently it is a burden to switch representations, not just because of the effort of defunctionalization, but because any changes to the search order may break existing tests, and make it difficult to compare benchmark results.

The rules for defunctionalization presented in Section 4.3, which preserve the exact search order, allow us to have the best of both worlds. We can have a higher-order implementation for pedagogical purposes, for example, and switch to a first-order implementation when it is time to debug or trace the code. And we can be confident that our existing tests, applications, and benchmark programs will not need to change. Of course, while we are living in the first-order world, we can easily change the search, or use an external process to guide the search.

We hope that there will be two positive outcomes to making it easier to switch between higher-order and first-order implementations of miniKanren. First, the wider availability of first-order implementations should make it much easier to produce the tools that the miniKanren ecosystem desperately needs, such as tracers, steppers, and debuggers. The rules in Section 4.3 imply that a programmer can debug a program in a first-order implementation of miniKanren, and be sure the program would show equivalent behavior in a higher-order miniKanren. Secondly, by decoupling the search from the program representation, it should be easier to experiment with novel search techniques, such as driving the search from an external process, or mixing different searches within a single miniKanren program. This should make it easier and faster to explore program synthesis and other advanced relational programming topics.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. Research reported in this publication was supported in part by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number 3OT2TR002517-01S1. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

REFERENCES

[1] 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), 8.

[6] https://github.com/michaelballantyne/mk-interp/blob/master/mk.rkt

[7] https://github.com/michaelballantyne/backjumping-miniKanren/blob/master/backjumping2-defun4.rkt

[2] Jason Hemann and Daniel P. Friedman. 2013. μKanren: A minimal functional core for relational programming. In *Scheme and Functional Programming Workshop 2013*.
[3] Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. 2019. Relational Interpreters for Search Problems. In *Proceedings of the 2019 miniKanren Workshop*.
[4] John C Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*. ACM, 717–740.
[5] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. 2019. Certified Semantics for miniKanren. In *Proceedings of the 2019 miniKanren Workshop*.
[6] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William E. Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. 2018. Neural guided constraint logic programming for program synthesis. In *Advances in Neural Information Processing Systems*. 1737–1746.

A STEPPER OUTPUT FOR APPENDO

This appendix shows the interactive stepper output for the query `(query (x y) (appendo x y '(1 2 3)))` with a correctly implemented appendo shown in Section 2. Compare this output with that of a buggy appendoh implementation in Figure 1

```
Stepping through query: (query (x y) (appendo x y '(1 2 3)))
================================================================================
Current Depth: 0             Number of Choices: 1

| Choice 1:
|  x = #s(var x 1)
|  y = #s(var y 2)
|  Constraints:
|  * (appendo #s(var x 1) #s(var y 2) (1 2 3))

[h]elp, [u]ndo, or choice number> 1
================================================================================
Current Depth: 1             Number of Choices: 2

| Choice 1:
|  x = ()
|  y = (1 2 3)
|  No constraints

| Choice 2:
|  x = (1 . #s(var b 4))
|  y = #s(var y 2)
|  Constraints:
|  * (appendo #s(var b 4) #s(var y 2) (2 3))

[h]elp, [u]ndo, or choice number> 2
================================================================================
Current Depth: 2             Number of Choices: 2

| Choice 1:
|  x = (1)
|  y = (2 3)
|  No constraints

| Choice 2:
|  x = (1 2 . #s(var b 7))
|  y = #s(var y 2)
|  Constraints:
|  * (appendo #s(var b 7) #s(var y 2) (3))

[h]elp, [u]ndo, or choice number>
```

B IMPLEMENTATION OF APPENDOH

This appendix shows the implementation of the buggy appendoh that generated the stepper output in Figure 1.

```
(define-relation (appendoh l s ls)
  (conde
    ((== '() l) (== s ls))
    ((fresh (a d res)
       (== `(,a . ,d) l)
       (== `(,a . ,res) ls)
       (appendoh d s ls)))))
```

C RECOVERING MINIKANREN

We show an implementation miniKanren on top of μKanren. This version of miniKanren is compatible with both the higher-order μKanren implemented in Section 4.2, and the first-order μKanren implemented in Section 4.3.

```
(define-syntax define-relation
  (syntax-rules ()
    ((_ (name param ...) g ...)
     (define (name param ...)
       (relate (lambda () (fresh () g ...)) `(,name name ,param ...))))))
;; Low-level goals
(define succeed (== #t #t))
(define fail    (== #f #t))
(define-syntax conj*
  (syntax-rules ()
    ((_)                succeed)
    ((_ g)              g)
    ((_ gs ... g-final) (conj (conj* gs ...) g-final))))
(define-syntax disj*
  (syntax-rules ()
    ((_)           fail)
    ((_ g)         g)
    ((_ g0 gs ...) (disj g0 (disj* gs ...)))))
;; High level goals
(define-syntax fresh
  (syntax-rules ()
    ((_ (x ...) g0 gs ...)
     (let ((x (var/fresh 'x)) ...) (conj* g0 gs ...)))))
(define-syntax conde
  (syntax-rules ()
    ((_ (g gs ...) (h hs ...) ...)
     (disj* (conj* g gs ...) (conj* h hs ...) ...))))
;; Queries
(define-syntax query
  (syntax-rules ()
    ((_ (x ...) g0 gs ...)
```

```
    (let ((goal (fresh (x ...) (== (list x ...) initial-var) g0 gs ...)))
      (pause empty-state goal)))))
(define (stream-take n s)
  (if (eqv? 0 n) '()
    (let ((s (mature s)))
      (if (pair? s)
        (cons (car s) (stream-take (and n (- n 1)) (cdr s)))
        '()))))
(define-syntax run
  (syntax-rules ()
    ((_ n body ...) (map reify/initial-var (stream-take n (query body ...))))))
(define-syntax run*
  (syntax-rules () ((_ body ...) (run #f body ...))))
```

D RAW APPENDO TRACE

This appendix shows how the search representation changes as we step through the query `(query (xs) (appendo xs xs '(a a` using interleaving search. We need to transform this representation to recover the easier-to-read textual representations in Figure 1 and Appendix A.

```
Step 0:
(pause
 (state ())
 (conj
  (== (#s(var xs 1)) #s(var #f 0))
  (relate (appendo #s(var xs 1) #s(var xs 1) (a a)))))

Step 1:
(mplus
 (bind #f (relate (appendo #s(var xs 1) #s(var xs 1) (a a))))
 (pause
  (state ((== #s(var #f 0) (#s(var xs 1)))))
  (disj
   (conj (== #s(var xs 1) ()) (== #s(var xs 1) (a a)))
   (conj
    (conj
     (== (#s(var a 2) . #s(var b 3)) #s(var xs 1))
     (== (#s(var a 2) . #s(var c 4)) (a a)))
    (relate (appendo #s(var b 3) #s(var xs 1) #s(var c 4)))))))

Step 2:
(pause
 (state ((== #s(var #f 0) (#s(var xs 1)))))
 (disj
  (conj (== #s(var xs 1) ()) (== #s(var xs 1) (a a)))
  (conj
   (conj
```

```
    (== (#s(var a 2) . #s(var b 3)) #s(var xs 1))
    (== (#s(var a 2) . #s(var c 4)) (a a)))
   (relate (appendo #s(var b 3) #s(var xs 1) #s(var c 4))))))

Step 3:
(mplus
 (pause
  (state ((== #s(var #f 0) (#s(var xs 1)))))
  (conj
   (conj
    (== (#s(var a 2) . #s(var b 3)) #s(var xs 1))
    (== (#s(var a 2) . #s(var c 4)) (a a)))
   (relate (appendo #s(var b 3) #s(var xs 1) #s(var c 4)))))
 (bind #f (== #s(var xs 1) (a a))))

Step 4:
(mplus
 (bind #f (== #s(var xs 1) (a a)))
 (mplus
  (bind
   (mplus (bind #f (== (#s(var a 2) . #s(var c 4)) (a a))) #f)
   (relate (appendo #s(var b 3) #s(var xs 1) #s(var c 4))))
  (pause
   (state
    ((== #s(var c 4) (a))
     (== #s(var a 2) a)
     (== #s(var xs 1) (a . #s(var b 3)))
     (== #s(var #f 0) ((a . #s(var b 3))))))
   (disj
    (conj (== #s(var b 3) ()) (== (a . #s(var b 3)) (a)))
    (conj
     (conj
      (== (#s(var a 5) . #s(var b 6)) #s(var b 3))
      (== (#s(var a 5) . #s(var c 7)) (a)))
     (relate (appendo #s(var b 6) (a . #s(var b 3)) #s(var c 7)))))))))

Step 5:
(mplus
 (bind
  (mplus (bind #f (== (#s(var a 2) . #s(var c 4)) (a a))) #f)
  (relate (appendo #s(var b 3) #s(var xs 1) #s(var c 4))))
 (pause
  (state
   ((== #s(var c 4) (a))
    (== #s(var a 2) a)
    (== #s(var xs 1) (a . #s(var b 3)))
```

```
    (== #s(var #f 0) ((a . #s(var b 3))))))
  (disj
   (conj (== #s(var b 3) ()) (== (a . #s(var b 3)) (a)))
   (conj
    (conj
     (== (#s(var a 5) . #s(var b 6)) #s(var b 3))
     (== (#s(var a 5) . #s(var c 7)) (a)))
    (relate (appendo #s(var b 6) (a . #s(var b 3)) #s(var c 7)))))))

Step 6:
(pause
 (state
  ((== #s(var c 4) (a))
   (== #s(var a 2) a)
   (== #s(var xs 1) (a . #s(var b 3)))
   (== #s(var #f 0) ((a . #s(var b 3))))))
 (disj
  (conj (== #s(var b 3) ()) (== (a . #s(var b 3)) (a)))
  (conj
   (conj
    (== (#s(var a 5) . #s(var b 6)) #s(var b 3))
    (== (#s(var a 5) . #s(var c 7)) (a)))
   (relate (appendo #s(var b 6) (a . #s(var b 3)) #s(var c 7))))))

Step 7:
Answer: (state ((== #s(var #f 0) ((a)))))
Remaining stream:
(mplus
 (pause
  (state
   ((== #s(var c 4) (a))
    (== #s(var a 2) a)
    (== #s(var xs 1) (a . #s(var b 3)))
    (== #s(var #f 0) ((a . #s(var b 3))))))
  (conj
   (conj
    (== (#s(var a 5) . #s(var b 6)) #s(var b 3))
    (== (#s(var a 5) . #s(var c 7)) (a)))
   (relate (appendo #s(var b 6) (a . #s(var b 3)) #s(var c 7)))))
 (mplus (bind #f (== #s(var xs 1) #s(var c 4))) #f))
```

E PRUNING

We show the code used to eliminate obviously failing goals from our stream.

```
(define (prune/stream s)
  (match s
```

```
    ((mplus s1 s2) (match (prune/stream s1)
                     (#f (prune/stream s2))
                     (s1 (match (prune/stream s2)
                           (#f s1)
                           (s2 (mplus s1 s2))))))
    ((bind s g)    (match (prune/stream s)
                     (#f          #f)
                     (`(,st . #f) (prune/goal st g))
                     ((pause st g1)
                      (match (prune/goal st g)
                        (#f           #f)
                        ((pause st g) (pause st (conj g1 g)))))
                     (s (match (prune/goal empty-state g)
                          (#f          #f)
                          ((pause _ _) (bind s g))))))
    ((pause st g)  (prune/goal st g))
    (`(,st . ,s)   `(,st . ,(prune/stream s)))
    (s             s)))

(define (prune/goal st g)
  (define (prune/term t) (walk* t (state-sub st)))
  (match g
    ((disj g1 g2)
     (match (prune/goal st g1)
       (#f (prune/goal st g2))
       ((pause st1 g1)
        (match (prune/goal st g2)
          (#f           (pause st1 g1))
          ((pause _ g2) (pause st (disj g1 g2)))))))
    ((conj g1 g2)
     (match (prune/goal st g1)
       (#f           #f)
       ((pause st g1) (match (prune/goal st g2)
                        (#f           #f)
                        ((pause st g2) (pause st (conj g1 g2)))))))
    ((relate thunk d) (pause st (relate thunk (prune/term d))))
    ((== t1 t2)
     (let ((t1 (prune/term t1)) (t2 (prune/term t2)))
       (match (unify t1 t2 st)
         (#f          #f)
         (`(,st . #f) (pause st (== t1 t2)))))))))
```

F TRANSFORMATION TO DISJUNCTIVE NORMAL FORM

We show the code used to transform our streams into Disjunctive Normal Form (DNF).

```
(define (dnf/stream s)
```

```
  (define (push-pause st g)
    (match g
      ((disj g1 g2) (mplus (push-pause st g1) (push-pause st g2)))
      (g            (pause st g))))
  (match s
    ((bind s g)
     (let loop1 ((s (dnf/stream s)) (g (dnf/goal g)))
       (define (loop2 s g)
         (match g
           ((disj ga gb) (mplus (loop2 s ga) (loop2 s gb)))
           (g            (bind s g))))
       (match s
         ((mplus sa sb) (mplus (loop1 sa g) (loop1 sb g)))
         (`(,st . ,s)   (mplus (push-pause st g) (loop1 s g)))
         (s             (loop2 s g)))))
    ((pause st g)  (push-pause st (dnf/goal g)))
    ((mplus s1 s2) (mplus (dnf/stream s1) (dnf/stream s2)))
    (`(,st . ,s)   `(,st . ,(dnf/stream s)))
    (s             s)))

(define (dnf/goal g)
  (match g
    ((conj g1 g2)
     (let loop1 ((g1 (dnf/goal g1)) (g2 (dnf/goal g2)))
       (define (loop2 g1 g2)
         (match g2
           ((disj g2a g2b) (disj (loop2 g1 g2a) (loop2 g1 g2b)))
           (g2             (conj g1 g2))))
       (match g1
         ((disj g1a g1b) (disj (loop1 g1a g2) (loop1 g1b g2)))
         (g1             (loop2 g1 g2)))))
    ((disj g1 g2) (disj (dnf/goal g1) (dnf/goal g2)))
    (g            g)))
```

Relational Interpreters for Search Problems*

PETR LOZOV, EKATERINA VERBITSKAIA, and DMITRY BOULYTCHEV, Saint Petersburg State University, Russia and JetBrains Research, Russia

We address the problem of constructing a solver for a certain search problem from its solution verifier. The main idea behind the approach we advocate is to consider a verifier as an interpreter which takes a data structure to search in as a program and a candidate solution as this program's input. As a result the interpreter returns "*true*" if the candidate solution satisfies all constraints and "*false*" otherwise. Being implemented in a relational language, a verifier becomes capable of finding a solution as well. We apply two techniques to make this scenario realistic: *relational conversion* and *supercompilation*. Relational conversion makes it possible to convert a first-order functional program into relational form, while supercompilation (in the form of conjunctive partial deduction (CPD)) — to optimize out redundant computations. We demonstrate our approach on a number of examples using a prototype tool for OCanren — an implementation of miniKanren for OCaml, — and discuss the results of evaluation.

CCS Concepts: • **Software and its engineering → Constraint and logic languages**; **Source code generation**;

Additional Key Words and Phrases: relational programming, relational interpreters, search problems

1 INTRODUCTION

Verifying a solution for a problem is much easier than finding one — this common wisdom can be confirmed by anyone who used both to learn and to teach. This observation can be justified by its theoretical applications, thus being more than informal knowledge. For example, let us have a language $\mathcal{L}$. If there is a predicate $V_{\mathcal{L}}$ such that

$$\forall \omega \ : \ \omega \in \mathcal{L} \iff \exists p_\omega \ : \ V_{\mathcal{L}}(\omega, p_\omega)$$

(with p_ω being of size, polynomial on ω) and we can recognize $V_{\mathcal{L}}$ in a polynomial time, then we call $\mathcal{L}$ to be in the class NP [Garey and Johnson 1990]. Here p_ω plays role of a justification (or proof) for the fact $\omega \in \mathcal{L}$. For example, if $\mathcal{L}$ is a language of all hamiltonian graphs, then $V_{\mathcal{L}}$ is a predicate which takes a graph ω and some path p_ω and verifies that p_ω is indeed a hamiltonial path in ω. The implementation of the predicate $V_{\mathcal{L}}$, however, tells us very little about the *search procedure* which would calculate p_ω as a function of ω. For the whole class of NP-complete problems no polynomial search procedures are known, and their existence at all is a long-standing problem in the complexity theory.

There is, however, a whole research area of *relational interpreters*, in which a very close problem is addressed. Given a language $\mathcal{L}$, its *interpreter* is a function $\texttt{eval}_{\mathcal{L}}$ which takes a program $p^{\mathcal{L}}$ in the language $\mathcal{L}$ and an input i and calculates some output such that

$$\texttt{eval}_{\mathcal{L}}(p^{\mathcal{L}}, i) = [\![p^{\mathcal{L}}]\!]_{\mathcal{L}}\,(i)$$

*This work was partially suppored by the grant 18-01-00380 from The Russian Foundation for Basic Research

Authors' address: Petr Lozov, lozov.peter@gmail.com; Ekaterina Verbitskaia, kajigor@gmail.com; Dmitry Boulytchev, dboulytchev@math.spbu.ru, Saint Petersburg State University, Russia , JetBrains Research, Russia.

miniKanren.org/workshop/2019/8-ART3

where $[\![\bullet]\!]_{\mathcal{L}}$ is the semantics of the language $\mathcal{L}$. In these terms, a verification predicate $V_{\mathcal{L}}$ can be considered as an interpreter which takes a program ω, its input p_ω and returns *true* or *false*. A *relational* interpreter is an interpreter which is implemented not as a function $\texttt{eval}_{\mathcal{L}}$, which calculates the output from a program and its input, but as a relation $\texttt{eval}^o_{\mathcal{L}}$ which connects a program with its input and output. This alone would not have much sense, but if we allow the arguments of $\texttt{eval}^o_{\mathcal{L}}$ to contain *variables* we can consider relational interpreter as a generic search procedure which determines the values for these variables making the relation hold. Thus, with relational interpreter it is possible not only to calculate the output from an input, but also to run a program in an opposite "direction", or to synthesize a program from an input-output pair, etc. In other words, relational verification predicate is capable (in theory) to both *verify* a solution and *search* for it.

Implementing relational interpreters amounts to writing it in a relational language. In principle, any conventional language for logic programming (Prolog [Clocksin and Mellish 2003], Mercury [Somogyi et al. 1996], etc.) would make the job. However, the abundance of extra-logical features and the incompleteness of default search strategy put a number of obstacles on the way. There is, however, a language specifically designed for pure relational programming, and, in a narrow sense, for implementing relational interpreters — miniKanren [Friedman et al. 2005]. Relational interpreters, implemented in miniKanren, demonstrate all their expected potential: they can synthesize programs by example, search for errors in partially defined programs [Byrd et al. 2017], produce self-evaluated programs [Byrd et al. 2012], etc. However, all these results are obtained for a family of closely related Scheme-like languages and require a careful implementation and even some *ad-hoc* optimizations in the relational engine.

From a theoretical standpoint a single relational interpreter for a Turing-complete language is sufficient: indeed, any other interpreter can be turned into a relational one just by implementing it in a language, for which relational interpreter already exists. However, the overhead of additional interpretation level can easily make this solution impractical. The standard way to tackle the problem is partial evaluation or specialization [Jones et al. 1993]. A *specializer* $\texttt{spec}_{\mathcal{M}}$ for a language $\mathcal{M}$ for any program $p^{\mathcal{M}}$ in this language and its partial input i returns some program which, being applied to the residual input x, works exactly as the original program on both i and x:

$$\forall x \ : \ [\![\texttt{spec}_{\mathcal{M}}\,(p^{\mathcal{M}}, i)]\!]_{\mathcal{M}}\,(x) = [\![p^{\mathcal{M}}]\!]_{\mathcal{M}}\,(i, x).$$

If we apply a specializer to an interpreter and a source program, we obtain what is called *the first Futamura projection* [Futamura 1971]:

$$\forall i \ : \ [\![\texttt{spec}_{\mathcal{M}}\,(\texttt{eval}^{\mathcal{M}}_{\mathcal{L}}, p^{\mathcal{L}})]\!]_{\mathcal{M}}\,(i) = [\![\texttt{eval}^{\mathcal{M}}_{\mathcal{L}}]\!]_{\mathcal{M}}\,(p^{\mathcal{L}}, i).$$

Here we added an upper index $\mathcal{M}$ to $\texttt{eval}_{\mathcal{L}}$ to indicate that we consider it as a program in the language $\mathcal{M}$. In other words, the first Futamura projection specializes an interpreter for a concrete program, delivering the implementation of this program in the language of interpreter implementation. An important property of a specializer is *Jones-optimality* [Jones et al. 1993], which holds when it is capable to completely eliminate the interpretation overhead in the first Futamura projection. In our case $\mathcal{M}$ = miniKanren, from which we can conclude that in order to eliminate the interpretation overhead we need a Jones-optimal specializer for miniKanren. Although implementing a Jones-optimal specializer is not an easy task even for simple functional languages, there is a Jones-optimal specializer for a logical language [Leuschel et al. 2004], but not for miniKanren.

The contribution of this paper is as follows:

- We demonstrate the applicability of relational programming and, in particular, relational interpreters for the task of turning verifiers into solvers.
- To obtain a relational verifier from a functional specification we apply *relational conversion* [Byrd 2009; Lozov et al. 2018] — a technique which for a first-order functional program directly constructs its relational counterpart. Thus, we introduce a number of new relational interpreters for concrete search problems.

- We employ supercompilation in the form of conjunctive partial deduction (CPD) [De Schreye et al. 1999] to eliminate the redundancy of a generic search algorithm caused by partial knowledge of its input.
- We give a number of examples and perform an evaluation of various solutions for the approach we address.

Both relational conversion and conjunctive partial deduction are done in an automatic manner. The only thing one needs to specify is the known arguments or the execution direction of a relation.

As concrete implementation of miniKanren we use OCanren [Kosarev and Boulytchev 2016] — its embedding in OCaml; we use OCaml to write functional verifiers; our prototype implementation of conjunctive partial deduction is written in Haskell.

The paper is organized as follows. In Section 2 we give a complete example of solving a concrete problem — searching for a path in a graph, — with relational verifier. Section 3 recalls the cornerstones of relational programming in miniKanren and the relational conversion technique. In Section 4 we describe how conjunctive partial deduction was adapted for relational programming. Section 5 presents the evaluation results for concrete solvers built using the technique in question. The final section concludes.

2 SEARCHING FOR PATHS IN A GRAPH WITH A RELATIONAL VERIFIER

In this section we demonstrate how to solve a concrete problem of searching for paths in a directed graph with a relational verifier. A directed graph is a tuple $(N, E, start, end)$, where N is a finite set of *nodes*, E is a finite set of *edges*, functions $start, end : E \to N$ return a start and an end nodes for a given edge respectively. A path in a directed graph is a sequence:

$$\langle n_0, e_0, n_1, e_1, \ldots, n_k, e_k, n_{k+1} \rangle$$

such that

$$\forall i \in \{0 \ldots k\} \ : \ n_i = start\,(e_i) \text{ and } n_{i+1} = end\,(e_i).$$

The problem of searching for paths in a graph is to find a set $\{p \mid p \text{ is a path in } g\}$, where g is a graph. There are many concrete algorithms which search for paths in a graph. Implementing any of them involves determining in which way to traverse the graph, how to ensure one does not get stuck exploring a cycle in the graph (a cycle is a path in the graph of form $\langle n_0, e_0, \ldots, n_k, e_k, n_0 \rangle$), how to ensure one path is not processed multiple times, and so on. A much easier task is to implement a simple verifier, which checks if a sequence is indeed a path in a graph, and generate the path searching routine from it by the relational conversion.

Below is the implementation of the verifier "`isPath`". This function takes as an input a list of nodes "`ns`" and a graph "`g`". We represent the graph as a list of edges, stipulating there are no parallel edges. Each edge e is represented as a pair of nodes (n, m), where $n = start(e)$, $m = end(e)$. Given $ns = [n_0, \ldots, n_{k+1}]$ and a graph $g = [e_0, \ldots, e_l]$, the function returns true, if $\exists i_0 \ldots i_k$ such that $\langle n_0, e_{i_0}, n_1, e_{i_1}, \ldots, e_{i_k}, n_{k+1} \rangle$ is a path in g.

```
1  let rec isPath ns g =
2    match ns with
3    | x1 :: x2 :: xs  → elem (x1, x2) g && isPath (x2 :: xs) g
4    | [_]             → true
```

The function "`elem`" checks if an edge "`e`" exists in the graph "`g`". We omit the definition of equality check for edges "`eq`", since it is trivial to implement and is not relevant for the example.

```
let rec elem e g =
  match g with
  | []      → false
  | x :: xs → if eq e x then true else elem e xs
```

We stipulate that a path must include at least two nodes, since searching for shorter paths is trivial. Line 3 of the "isPath" definition checks that the first two nodes of the list form an edge of the graph. Then it checks that what is left after deleting the first node from the list is still a path in the graph. Line 4 may come off a little counterintuitive, since it states that a path which includes a single arbitrary node is in the input graph. However we only execute this branch by a recursive call of "isPath", which only happens after we have already ensured with the call to the "elem" function that the said node is in the graph.

The relational conversion of the verifier function "isPath" generates a relation "$\texttt{isPath}^o$" defined for a path "ns", a graph "g" and a boolean value "res", which is true if "ns" is a path in the graph "g" and false otherwise. The function "elem" is transformed into a relation "$\texttt{elem}^o$" defined for an edge "e", a graph "g" and a boolean value "res", which is true if "e" is an edge in the graph "g" and false otherwise. The result of the relational conversion of the functions "isPath" and "elem" is presented below.

```
5  let rec elem^o e g res = conde [
6    (g ≡ nil () ∧ res ≡ ↑false);
7    (fresh (x xs resEq) (
8      (g ≡ x % xs) ∧
9      (eq^o e x resEq) ∧
10     (conde [
11       (resEq ≡ ↑true  ∧ res ≡ ↑true);
12       (resEq ≡ ↑false ∧ elem^o e xs res)])))]
13
14 let rec isPath^o ns g res = conde [
15   (fresh (el) (
16     (ns ≡ el % nil ()) ∧
17     (res ≡ ↑true));
18  (fresh (x1 x2 xs resElem resIsPath) (
19     (ns ≡ x1 % (x2 % xs)) ∧
20     (elem^o (pair x1 x2) g resElem) ∧
21     (isPath^o (x2 % xs) g resIsPath) ∧
22     (conde [
23       (resElem ≡ ↑false ∧ res ≡ ↑false);
24       (resElem ≡ ↑true  ∧ res ≡ resIsPath)])))]
```

Here we use the syntax of OCanren. A new relation is defined as a recursive function with the keywords "**let rec**". The body of the relation is a goal created with the following goal constructors.

- Disjunction $g_1 \vee g_2$, where g_1, g_2 — some goals. The two goals are evaluated independently and their results are combined.
- Disjunction of goal list **conde** $[g_1; \ldots; g_n]$, where $g_1; \ldots; g_n$ — some goals.
- Conjunction $g_1 \wedge g_2$, where g_1, g_2 — some goals. The goal g_2 is evaluated only if the evaluation of g_1 succeeded; the evaluation of g_2 uses the results of g_1.
- Syntactic unification $t_1 \equiv t_2$, where t_1, t_2 — some terms. Unification is a basic goal constructor. If t_1 and t_2 can be unified, the goal is considered successful and failed otherwise.
- Relation call $r^n t_1 \ldots t_n$ where r^n is a name of some n-ary relation, and t_i are terms.
- To introduce fresh variables into scope, one should use **fresh** $(\overline{x})$ g, where $\overline{x}$ is a list of variable names.

Besides goal constructors we use some syntactic sugar for values and lists. "↑" is used to transform a value into a logic value. The empty list is represented as "`nil ()`", and to construct a new list from a value "`h`" and a list "`t`" we use "`h % t`". A tuple of "`x`" and "`y`" is created with "`pair x y`".

Regrettably, this relational interpreter suffers from poor performance. Query "`isPath`^{o} `q <graph>` **true**" for path searching takes more than ten minutes even for graphs of 5 nodes. This is somewhat expected, considering that the relational conversion generates a relation which can be used for many different queries, which is excessive when any particular query is in question. This is, of course, not a desirable behaviour. Fortunately, further transformation of the relation can improve the performance.

For example, if we consider a query "`isPath`^{o} `q <graph>` ↑**true**", we can simplify lines 18 through 24 of its definition. First, we notice that, having "`res`" be equal to "↑**true**", we can safely remove the disjunct in line 23, after what the whole "**conde**" becomes unnecessary and can be removed. After moving the unifications for "`resElem`" and "`resIsPath`" to the top level, we get the following equivalent definition of the "`isPath`^{o}" relation. Note, that the call to the "`elem`^{o}" relation is done with the last argument being unified with "↑**true**", so further specialization is still possible.

```
25 let rec isPath^o ns g res = conde [
26   (fresh (el) (
27     (ns ≡ el % nil ()) ∧
28     (res ≡ ↑true)));
29   (fresh (x1 x2 xs resElem resIsPath) (
30     (resElem ≡ ↑true) ∧
31     (resIsPath ≡ ↑true) ∧
32     (ns ≡ x1 % (x2 % xs)) ∧
33     (elem^o (pair x1 x2) g resElem) ∧
34     (isPath^o (x2 % xs) g resIsPath)))]
```

The specialized version of the relation is much more performant than the original one. Before, searching paths of length 5 took more than 10 minutes while the specialized version finds paths of length 10 in the graph with 100 edges in a few seconds.

This transformation can be performed automatically with conjunctive partial deduction. The result of partially deducing the "`isPath`^{o} `q p` ↑**true**", where "`p`" and "`q`" are fresh variables is about 40 lines of code long and it has the same performance as the manually transformed relation. We omit the transformed program because of the space concerns, but it can be found in the repository[1].

3 RELATIONAL CONVERSION

In this section we describe how the relational conversion in the form of *unnesting* [Byrd 2009] is done. Unnesting constructs a relational program by a first-order functional program.

First, a new variable for every subexpression is introduced with the **let**-expression. Then, all pattern matching and if-expressions are translated into disjunctions, in which unifications are generated for the patterns. Free variables are introduced into scope with the **fresh**. Every n-ary function becomes $(n + 1)$-ary relation with the last argument unified with the result. As a final step, unifications are reordered with relation calls such that to be computed as early as it is possible.

The example of unnesting is shown in Fig. 1. The input functional program is presented in Fig. 1a. The result of introducing fresh variables for subexpressions is in Fig. 1b. The relational program before the conjuncts are reordered is shown in Fig. 1c and the result of the unnesting is presented in Fig. 1d.

[1] https://github.com/Lozov-Petr/miniKanren-2019-Relational-Interpreters-for-Search-Problems

```
let rec append a b =
  match a with
  | []      → b
  | x :: xs →
    x :: append xs b
```

(a)

```
let rec append a b =
  match a with
  | []      → b
  | x :: xs →
    let q = append xs b in
    x :: q
```

(b)

```
let rec append^o a b c =
  (a ≡ [] ∧ b ≡ c) ∨
  (fresh (x xs q) (
     (a ≡ x :: xs) ∧
     (append^o xs b q) ∧
     (c ≡ x :: q))
```

(c)

```
let rec append^o a b c =
  (a ≡ [] ∧ b ≡ c) ∨
  (fresh (x xs q) (
     (a ≡ x :: xs) ∧
     (c ≡ x :: q) ∧
     (append^o xs b q))
```

(d)

Fig. 1. Example of unnesting

Note, that the unnesting has limitations: it does not support higher-order functions and partial application. A more general method of translation which does not impose the same limitations was developed [Lozov et al. 2018]. Unfortunately, it uses higher-order relations which are not currently supported in conjunctive partial deduction, so we use unnesting.

The forward execution of the relation mimics the execution of the function from which it was constructed by relational conversion. This makes forward execution quite efficient, to the detriment of the execution in the backwards direction. The unnesting can be modified to improve the performance of backward execution. Let us consider the conversion of a functional conjunction "$f_1\ x_1$ && $f_2\ x_2$".

```
λ res →
  fresh (p) (
    (f1 x1 p) ∧
    (conde [
      (p ≡ ↑false ∧ res ≡ ↑false);
      (p ≡ ↑true  ∧ f2 x2 res)]))
```

Mimicking the function evaluation, the forward execution of this code first computes "$f_1\ x_1$". If it fails, then the result "res" is unified with "**false**", otherwise the second conjunct "$f_2\ x_2$" is executed and its result is unified with the result. This strategy is not efficient in the backward direction, when we know what "res" is. The following relation is much more performant when executed in the backward direction:

```
λ res →
    conde [
      (res ≡ ↑false ∧ f1 x1 ↑false);
      (f1 x1 ↑true  ∧ f2 x2 res)]
```

In particular, if "res ≡ ↑**true**", both conversions execute "$f_2\ x_2$ res", but when the first conversion computes "$f_1\ x_1$ p" with fresh "p", the second executes "$f_1\ x_1$ ↑**true**". Using the second conversion is enough to significantly

increase the performance in the backward direction. For example, the path search takes several minutes if the first conversion strategy is used, whereas it finishes in less than a second in the second case.

Choosing the second conversion strategy comes with a price for the forward execution. Instead of executing "$f_1\ x_1\ p$", where "p" is fresh, the second strategy executes both "$f_1\ x_1 \uparrow$**false**" and "$f_1\ x_1 \uparrow$**true**". In the worst case scenario, when the execution of "f_1" does not depend on the last argument, it doubles the number of executions of "f_1".

To sum up, by choosing different strategies of the relational conversion we can achieve significant performance improvement. There is no single right way of doing the conversion which improves the performance of the execution in every possible direction. Choosing a strategy per each relation and each direction manually is not feasible, but it can be achieved with a fully-automatic program transformation, such as conjunctive partial deduction.

4 CONJUNCTIVE PARTIAL DEDUCTION

Specialization [Jones et al. 1993] is a natural way to tackle the problem of redundant computations when a part of the input is known. A fully-automatic specialization technique developed in the domain of logic programming is called *partial deduction* [Komorowski 1982; Lloyd and Shepherdson 1991]. It is related to the supercompilation of functional languages [Glück and Sørensen 1994; Turchin 1986]. The particular flavour of the partial deduction we are interested in is called *conjunctive partial deduction* [De Schreye et al. 1999]. As opposed to the partial deduction, conjunctive partial deduction handles conjunctions of atoms, thus being able to perform such optimizations as tupling [Hu et al. 1997] and deforestation [Wadler 1988]. Below we demonstrate by example the features of conjunctive partial deduction.

Deforestation is a program transformation which gets rid of intermediate data structures. The following example demonstrates deforestation. Consider a goal "append^o xs ys ts $\wedge$ append^o ts zs rs" (note the shared "ts"), where "append^o x y xy" describes concatenation, "nil ()" constructs the empty list, and "h % t" constructs a new list from the value "h" and another list "t" (similarly to "cons" in Scheme and " :: " in OCaml).

```
let rec appendo x y xy = conde [
  (x ≡ nil () ∧ xy ≡ y);
  (fresh (h t ty) (
     (x  ≡ h % t)  ∧
     (xy ≡ h % ty) ∧
     (appendo t y ty)))]
```

This goal concatenates three lists: "xs", "ys", "zs", constructing an intermediate list "ts". During the execution of this goal, elements of the list "xs" are examined twice: first when "ts" is constructed, and then when the result "rs" is constructed. What is worse, "ts" is only constructed to be immediately deconstructed. Deforestation gets rid of "ts" in this example.

A better program would be such that does not construct "ts" at all. Such a program be generated from the original definition by conjunctive partial deduction and is shown below:

```
let rec doubleAppendo xs ys zs rs = conde [
  (xs ≡ nil () ∧ appendo ys zs rs);
  (fresh (h t ts) (
     (xs ≡ h % t)  ∧
     (rs ≡ h % ts) ∧
     (doubleAppendo t ys zs ts)))]
```

Conjunctive partial deduction is also capable of *tupling*. This transformation makes sure that the same data structure is traversed once even if computing several results. The following example demonstrates such a case.

The goal "`maxLength`^{o} `xs m l`" computes both the maximum value of the list "`xs`" and its length. The elements of the list are Peano numbers with "`zero ()`" as the zero and "`succ`" as the successor function. The third argument "`b`" of the relation "`le`^{o} `x y b`" is "$\uparrow$**true**" if "`x`" is less or equal than "`y`", and "$\uparrow$ **false** " otherwise. The relation "`gt`^{o} `x y b`" is similar to "`le`^{o} `x y b`", but it checks for "`x`" to be greater than "`y`".

```
let maxLength^o xs m l = max^o xs m ∧ length^o xs l

let rec length^o xs l = conde [
  (xs ≡ nil () ∧ l ≡ zero ());
  (fresh (h t m) (
    xs ≡ h % t ∧ l ≡ succ m ∧ length^o t m))]

let max^o xs m = max^o_1 xs (zero ()) m

let rec max^o_1 xs n m = conde [
  (xs ≡ nil () ∧ m ≡ n);
  (fresh (h t) (
    (xs ≡ h % t) ∧
    (conde [
      (le^o h n ↑true ∧ max^o_1 t n m);
      (gt^o h n ↑true ∧ max^o_1 t h m)])))]

let rec le^o x y b = conde [
  (x ≡ zero () ∧ b ≡ ↑true);
  (fresh (x_1) (
    x ≡ succ x_1 ∧ y ≡ zero () ∧ b ≡ ↑false));
  (fresh (x_1 y_1) (
    x ≡ succ x_1 ∧ y ≡ succ y_1 ∧ le^o x_1 y_1 b))]

let rec gt^o x y b = conde [
  (x ≡ zero () ∧ b ≡ ↑false);
  (fresh (x_1) (
    x ≡ succ x_1 ∧ y ≡ zero () ∧ b ≡ ↑false));
  (fresh (x_1 y_1) (
    x ≡ succ x_1 ∧ y ≡ succ y_1 ∧ gt^o x_1 y_1 b))]
```

Execution of the goal "`maxLength`^{o} `xs m l`" leads to "`xs`" being traversed twice. There is a way to rewrite the program so that "`xs`" is traversed once, but this requires fusing together the definitions of "`length`^{o}" and "`max`^{o}", which either restricts code reuse, or leads to code duplication. A better way is to only fuse the definitions when it is needed, and do it automatically by employing tupling.

The desirable implementation of the "`maxLength`^{o} `xs m l`" relation is the following (the definitions of "`gt`^{o}" and "`le`^{o}" are left out for brevity). It can be achieved with conjunctive partial deduction as well:

```
let maxLength^o xs m l = maxLength_1^o xs m (zero ()) l

let rec maxLength_1^o xs m n l = conde [
  (xs ≡ nil () ∧ m ≡ n ∧ l ≡ zero ());
  (fresh (h t l_1)
     (xs ≡ h % t) ∧
     (l ≡ succ l_1) ∧
     (conde [
       (le^o h n ∧ maxLength_1^o t m n l);
       (gt^o h n ∧ maxLength_1^o t m h l)]))]
```

4.1 CPD for Prolog-like languages

Initially, conjunctive partial deduction was developed for Prolog-like languages. Conjunctive partial deduction partially evaluates goals, which are conjunctions of atoms, using two levels of control: local and global [Glück et al. 1996]. The global control determines which atoms are to be partially deduced. The local control — what the definitions for the atoms selected at the global control shall be. Both local and global control construct tree structures which represent the input program.

Local control constructs finite SLD-trees for conjunctions of atoms. The construction is guided with an *unfold* operator: it selects a literal from the leaf of the partially constructed SLD-tree and adds its resolvents as children at each step. Since, in general, SLD-trees are infinite, a decision to stop unfolding should be made at some point. There are several techniques for doing this, the most promising of them combine determinacy and either some well-founded or well-quasi order, such as homeomorphic embedding, or other measures.

Global control determines the set of the conjunctions for which partial SLD-trees are built. The important goal of the global control is to ensure termination. The termination is achieved with the *abstraction*. If there is a goal which is embedded into the current goal, it points to the possibility of nontermination. The embedding tells that there is a certain similarity between the two goals, and if a current goal keeps being processed, then their similar subpart will appear again and again, causing nontermination. Whenever the embedding goals are detected, the current goal is abstracted to remove the common subgoal from consideration.

When the partial deduction is done, the only thing left is to construct the *residual program*. The clauses are generated from a partial SLD-tree, one tree per conjunction at the global level. A conjunction is uniquely *renamed* to give a name for the predicate being defined. All free variables of the root of the tree become arguments of the predicate. For each non-failing path in the SLD-tree a clause is generated: a substitution collected along the path is substituted into the head of the clause, and the body is generated from what is in the leaf.

4.2 CPD for miniKanren

In this section we describe how we adapted conjunctive partial deduction for miniKanren. We describe the particular unfolding and generalization strategies as well as discuss how the conjunctive partial deduction had to be modified as a response to the differences between Prolog and miniKanren.

4.2.1 Local Control. Goals in miniKanren are different from those in Prolog-like languages: besides conjunction, disjunction and relation calls, there are explicit unification and introduction of fresh variables. We normalize the input goal so that it was a disjunction of conjunctions of relation calls. To do so, we first pop all the fresh variables to the top level ("`fresh (x) (p (x) ∧ fresh (y) (q(x) ∨ r(y, x)))`" becomes "`fresh (x y) (p(x) ∨ q(x) ∧ r(y, x))`"). Then we transform the goal to be a disjuction of conjunctions of relation calls or unifications. All unifications in each conjunction are evaluated to some substitution (or the

conjunct is discarded, if some unification fails). The normalization allows us to only consider conjunctions of relation calls while doing conjunctive partial deduction.

The local control constructs the following tree structure which represents the goal:

```
type local_tree =
    Fail
  | Success of subst
  | Leaf    of goal list * subst
  | Disj    of local_tree list
  | Conj    of local_tree * goal list
```

Leaf nodes can be either "`Fail`", "`Success`" or "`Leaf`". The "`Fail`" node is created whenever the evaluation of the current goal fails. When the current goal evaluates to some substitution, we create the "`Success`" node with this substitution. The last leaf node is called "`Leaf`", it corresponds to some partially evaluated goal. This type of node contains a substitution which has been computed up to this point, and a residual goal. The goal in this type of node is then examined at the global level.

"`Disj`" node corresponds to a disjunction in a goal: its children are the local control trees constructed for all disjuncts. The last type of nodes is a "`Conj`" node. It is a transient node, which keeps track of a conjunction being unfolded.

In general, unfolding replaces some of the relation calls with their bodies and partially evaluates them. The particular unfolding strategy we adhere to is the following. At each step only one relation call is replaced with its body: the leftmost selectable relation call. The selectable relation call is the one which does not embed any of its predecessors — goals which were unfolded in order to get the current goal. Embedding here is the modification of the homeomorphic embedding defined for the conjunctions of goals in conjunctive partial deduction literature [De Schreye et al. 1999]. Since using pure embedding to control unfolding leads to hideously big programs, we also allow only one non-deterministic unfold.

4.2.2 Global Control. The conjunctions in the "`Leaf`" nodes are processed at the global level. This step is responsible for the termination of the transformation. Generally speaking, the danger for nontermination arises whenever we encounter a subgoal which we have encountered before: processing the same thing will lead to itself over and over again. To break the vicious circle, one needs to stop unfolding the encountered subgoal, this is what *abstraction* serves for.

The simplest case here is when we come upon the goal which is equal up to variable renaming to any other goal at the global level. When this happens, we stop exploring the goal. This is called *variant check* in the literature, and it is done both at the global and the local control levels.

The more complicated case is when a subpart of the goal repeats. This case we test with the modification of the homeomorphic embedding relation (strict homeomorphic embedding), initially developed for conjunctions. A conjunction $\overline{A}$ is considered embedded into a conjunction $\overline{B}$ when there is an ordered subconjunction within $\overline{A}$, each conjunct of which is embedded into the corresponding conjunct of $\overline{B}$:

$$\overline{A} = A_0 \wedge A_1 \wedge \cdots \wedge A_n \trianglelefteq B_0 \wedge B_1 \wedge \cdots \wedge B_m = \overline{B},\ \textit{if}\ \exists\{i_0 \ldots i_m \mid \forall j.i_j < i_{j+1}\} : \forall j \in \{0 \ldots m\}.A_{i_j} \trianglelefteq B_j$$

A single conjunct is embedded into another ($A_i \trianglelefteq B_j$) when the following relation holds and A_i is *not* a strict instance of the second one B_j:

$$X \trianglelefteq Y, \text{where } X \text{ and } Y \text{ are variables}$$

$$f(x_0, x_1, \ldots, x_n) \trianglelefteq f(y_0, y_1, \ldots, y_n) \Leftrightarrow \forall i \in \{0 \ldots n\}.x_i \trianglelefteq y_i$$

$$f \trianglelefteq g(y_0, y_1, \ldots y_m) \Leftrightarrow \exists i \in \{0 \ldots m\}.f \trianglelefteq y_i$$

This check determines two major causes of the growth within the conjunctions. The conjunction can grow in some argument of a relation call or the number of conjuncts itself can grow. To mitigate the first source of the growth, the bigger conjunction can be replaced with a *most specific generalization* of the two conjunctions. Otherwise we need to *split* the embedded subconjunction from the rest and start processing them separately. This process called *abstraction* removes the subconjunctions which cause potential nontermination, and what is left should indeed be processed further.

4.2.3 Residualization. After the transformation is finished, a *residual* program is constructed from the global control tree. A relation definition is generated for each conjunction at the global level (this is done with the renaming step of the original conjunctive partial deduction). First, a unique name is given for each conjunction. Then free variables of the conjunction are collected to become the arguments of the relation: the constructors and constants are omitted (for example "`f x (succ y)` ∧ `g (zero ()) z`" becomes "`fG x y z`". The body of the definition is generated from the local control tree which corresponds to the conjunction under consideration. The body is formed as a disjunction of conjunctions for the non-failure nodes of the local control tree. A computed substitution is transformed into a conjunction of unifications. Suitable definitions are chosen for a goal in a leaf, and the conjunction of their applications is generated. As a final step we perform redundant argument filtering as described in [Leuschel and Sørensen 1996], and introduce fresh variables where necessary.

5 EVALUATION

In this section we present an evaluation of the proposed approach. We have implemented several relational interpreters for different search problems which can be found in the repository mentioned before. Some of the simpler interpreters demonstrate good performance for different directions on their own and for them CPD transformation is not needed. Thus, we will focus on two search problems which are more complex: searching for a path in a graph and searching for a unifier [Baader and Snyder 2001] of two terms. For each problem we compare four programs.

(1) The solver generated by the unnesting alone.
(2) The solver generated by the unnesting strategy aimed at backward execution.
(3) The solver generated by the unnesting and then specialized by conjunctive partial deduction for the backward direction.
(4) The interpretation of the functional verifier with the relational interpreter implemented in Scheme [Byrd et al. 2017].

First, let us compare the performance of the solvers for the path searching problem. The implementation of the functional verifier for this problem is described in Section 2. We ran the search on a graph with 20 nodes and 30 edges, consequentially searching for paths of the length 5, 7, 9, 11, 13, and 15. We averaged the execution times over 10 runs of the same query. We the limited the execution time by 300 seconds, and if the execution time of some query exceeded the timeout, we put ">300s" in the result table and did not request the execution of queries for longer paths. The results are presented in Table 1.

We can conclude that the execution time increases with the length of the path to search, which is expected, since with the length of the path the number of the subpaths to be explored is increasing as well. The solver generated by the unnesting alone and the interpretation with the relational intepreter demonstrate poor performance. The first one is due to its inherently inefficient execution in backward direction, while the second suffers from the interpretation overhead. Both the unnesting aimed at the backward execution and the solver automatically transformed with conjunctive partial deduction show good performance. Conjunctive partial deduction performs more thorough specialization, thus producing more efficient program.

Now let us consider the problem of finding a unifier of two terms which have free variables. A term is either a variable $(X, Y, \dots)$ or some constructor applied to terms $(nil, cons(H, T), \dots)$. A substitution

Path length	5	7	9	11	13	15
Only conversion	0.01s	1.39s	82.13s	>300s	—	—
Backward oriented conversion	0.01s	0.37s	2.68s	2.91s	4.88s	10.63s
Conversion and CPD	0.01s	0.06s	0.34s	2.66s	3.65s	6.22s
Scheme interpreter	0.80s	8.22s	88.14s	191.44s	>300s	—

Table 1. Searching for paths in the graph

Terms	f(X, a)	f(a % b % nil, c % d % nil, L)	f(X, X, g(Z, t))
	f(a, X)	f(X % XS, YS, X % ZS)	f(g(p, L), Y, Y)
Only conversion	0.01s	>300s	>300s
Backward oriented conversion	0.01s	0.11s	2.26s
Conversion and CPD	0.01s	0.07s	0.90s
Scheme interpreter	0.04s	5.15s	>300s

Table 2. Searching for a unifier of two terms

maps a variable to a term. A unifier of two terms t and s is a substitution σ which equalizes them: $t\sigma = s\sigma$ by simultaneously substituting the variables for their images. For example, a unifier of the terms $cons(42, X)$ and $cons(Y, cons(Y, nil))$ is a substitution $\{X \mapsto cons(42, nil), Y \mapsto 42\}$.

We implemented a functional verifier which checks if a substitution equalizes two input terms. We represent a variable name as a unique Peano number. A substitution is represented as a list of terms, in which the index of the term is equal to the variable name to which the term is bound, so the substitution $\{X \mapsto cons(42, nil), Y \mapsto 42\}$ is represented as a list "`[cons (42, nil); 42]`". The verifier returns true if the input terms can be unified with the candidate substitution and false otherwise.

As in the previous problem, we compare four solvers generated for the verifier described. With each solver, we search for a unifier of two terms and compare the execution times. The time comparison is presented in Table 2. The first two rows of each column contain two terms being unified. We use uppercase letters from the end of the alphabet ($X, Y, \ldots$) to denote variables, lowercase letters from the beginning of the alphabet ($a, b, \ldots$) to denote constants (constructors with zero arguments), identifiers which start from the lowercase letter ($f, g, \ldots$) to denote constructors.

Note, we compute a unifier for two terms, but not necessarily the most general unifier. We can implement the most general unification in miniKanren, but achieving the comparable performance using relational verifiers requires additional check that the unifier is indeed the most general. We are currently working on the implementation of such relational verifier.

Here four solvers compare to each other similarly to the previous problem: unnesting demonstrates the worst execution time, relational interpretation in Scheme is a little better, while unnesting aimed at backward execution and conjunctive partial deduction significantly improve the performance.

There exist pairs of terms, for which either of the solvers fails to compute a unifier under 300 seconds. The example of such terms is "`f(A,B,C,A,B,C,D)`" and "`f(B,C,D,x(R,S),x(a,T),x(Q,b),x(a,b))`". This is caused by how general and declarative the verifier is: there is nothing in it to restrict the search space. We can modify the verifier with the additional check to ensure that there are no bound variables in the candidate unifier. This modification restricts the search space when there are many variables in the input terms. But it also changes the semantics of the initial verifier and, as a consequence, the solvers: only idempotent unifiers can be found.

To sum up, we demonstrated by two examples that it is possible to create problem solvers from verifiers by using relational conversion and conjunctive partial deduction. Currently conjunctive partial deduction improves the performance the most, as compared to interpreting verifiers with Scheme relational interpreter or doing relational conversion which is solely aimed at backward or forward execution.

6 CONCLUSION AND FUTURE WORK

We have presented a way to construct a solver for a search problem from its solution verifier by first doing a relational conversion and then specializing the relation according to the desired execution direction by means of conjunctive partial deduction.

There are a few directions for future work.

Even if we generate a relation optimized for the particular direction, executing it with miniKanren still carries some overhead of interpretation. We believe that the best performance can be achieved by generating a functional program from the relation optimized for the particular direction. This way we can avoid interpretation overhead but still get the benefits of the approach. The second direction is to explore other specialization techniques besides conjunctive partial deduction which are better suited for miniKanren programs.

REFERENCES

Franz Baader and Wayne Snyder. 2001. Handbook of Automated Reasoning. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, Chapter Unification Theory.

William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* Indiana University, Bloomington.

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 8 (Aug. 2017), 26 pages. https://doi.org/10.1145/3110252

William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). *Workshop on Scheme and Functional Programming* (2012).

William F. Clocksin and Christopher S. Mellish. 2003. *Programming in Prolog* (5 ed.). Springer, Berlin. https://doi.org/10.1007/978-3-642-55481-0

Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. 1999. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The Journal of Logic Programming* 41, 2-3 (1999), 231–277.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer.* The MIT Press.

Yoshihiko Futamura. 1971. Partial evaluation of ccomputation process-an approach to a compiler-compiler. *Systems, Compurters, Controls* 25 (1971), 45–50.

Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA.

Robert Glück, Jesper Jørgensen, Bern Martens, and Morten Heine Sørensen. 1996. Controlling conjunctive partial deduction. In *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 152–166.

Robert Glück and Morten Heine Sørensen. 1994. Partial deduction and driving are equivalent. In *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 165–181.

Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. 1997. Tupling calculation eliminates multiple data traversals. *ACM Sigplan Notices* 32, 8 (1997), 164–175.

Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Peter Sestoft.

H Jan Komorowski. 1982. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 255–267.

Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. *ACM SIGPLAN Workshop on ML* (2016).

Michael Leuschel, Stephen J Craig, Maurice Bruynooghe, and Wim Vanhoof. 2004. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*. Springer, 340–375.

Michael Leuschel and Morten Heine Sørensen. 1996. Redundant argument filtering of logic programs. In *International Workshop on Logic Programming Synthesis and Transformation*. Springer, 83–103.

John W. Lloyd and John C Shepherdson. 1991. Partial evaluation in logic programming. *The Journal of Logic Programming* 11, 3-4 (1991), 217–242.

Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2018. Typed Relational Conversion. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 39–58.

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming* 29, 1-3 (1996), 17–64.

Valentin F Turchin. 1986. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 3 (1986), 292–325.

Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*. Springer, 344–358.

Constructive Negation for MiniKanren

EVGENII MOISEENKO, Saint Petersburg State University and JetBrains Research, Russia

We present an extension of MiniKanren with the negation operator based on the method of *constructive negation*. The idea of this method is to constructively build a stream of answers for the negated goal by collecting and negating individual answers to the positive version of the goal. As we demonstrate on the series of examples constructive negation suits to pure logical nature of MiniKanren: the relations involving the negation operator still can be "run" in various directions.

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

Additional Key Words and Phrases: relational programming, constructive negation, OCanren

1 INTRODUCTION

MiniKanren [Byrd 2010; Friedman et al. 2005] is a minimalistic domain-specific language which brings features of logic programming into a host language. Typical MiniKanren implementation introduces only a few operators: conjunction, disjunction, unification (which can be seen as equality constraint) and fresh variable introduction (existential quantification). Although this basis constitutes a Turing-complete language, in practice there are some cases when the availability of negative reasoning is desirable.

For example, consider the following problem. Suppose we want to write a program which removes the first occurrence of a given element from a list. In order to do that in MiniKanren we have to first define a ternary relation `remove` which binds the desired element, original list, and the same list after the deletion. Substituting the first argument of the relation with some element `e`, the second argument with some list `xs` and the third argument with a free variable `q` gives us a *goal* which is a proof search procedure. When passed to the `run` function, the goal will produce a lazy stream of *answers*. Each answer is represented by substitution which binds free variables to some terms. In the case of `remove`, we would expect a single answer, which binds `q` to the list, equal to `xs`, except that the first occurrence of `e` is removed.

The code on Listing 1 demonstrates a possible implementation of `remove`. It consists of three disjuncts, which represent three different cases. First, if the original list is empty, then the resulting list should also be empty. If the original list is not empty and its head is equal to the given element, then the resulting list should be equal to the tail. Finally, in the case when the head is not equal to the given element, the resulting list should be equal to the original, from the tail of which the occurrence of the element `e` is deleted.

Given the definition of `remove` from Listing 1, the following query `run (remove 2 [1;2;3] q)` will wrongly return two answers: `q =[1;3]` and `q =[1;2;3]` . The redundant (and, indeed, incorrect) answer `q =[1;2;3]` arose because of the third disjunct from `remove` definition, which always succeeds and thus generates a copy of the original list. We can prevent this behavior using *disequality constraint*, yet another primitive, which some MiniKanren implementations provide. Adding constraint `x ≠ e` to the third case makes all disjuncts mutually

Author's address: Evgenii Moiseenko, e.moiseenko@2012.spbu.ru, Saint Petersburg State University, Saint Petersburg , JetBrains Research, Russia.

miniKanren.org/workshop/2019/8-ART4

disjoint. With the fixed definition of `remove` (Listing 2), the query given above returns the single answer `q` =[1;3] as expected.

```
let remove e xs ys =
  (xs ≡ [] ∧ ys ≡ [])
  ∨
  fresh (x xs') (
    x ≡ e ∧
    xs ≡ x::xs' ∧
    ys ≡ xs'
  ) ∨
  fresh (x xs' ys') (
    xs ≡ x::xs' ∧
    ys ≡ x::ys' ∧
    remove e xs' ys'
  )
```

Listing 1. A flawed definition of `remove` relation

```
let remove e xs ys =
  (xs ≡ [] ∧ ys ≡ [])
  ∨
  fresh (x xs') (
    x ≡ e ∧
    xs ≡ x::xs' ∧
    ys ≡ xs'
  ) ∨
  fresh (x xs' ys') (
    x ≢ e ∧
    xs ≡ x::xs' ∧
    ys ≡ x::ys' ∧
    remove e xs' ys'
  )
```

Listing 2. The correct definition of the `remove` relation

```
let remove p xs ys =
  (xs ≡ [] ∧ ys ≡ [])
  ∨
  fresh (x xs') (
    p x ∧
    xs ≡ x::xs' ∧
    ys ≡ xs'
  ) ∨
  fresh (x xs' ys') (
    ¬(p x) ∧
    xs ≡ x::xs' ∧
    ys ≡ x::ys' ∧
    remove p xs' ys'
  )
```

Listing 3. A generalized version of the `remove` relation

A disequality constraint represents a very limited form of negation, which is often not sufficient. Imagine that we want to generalize `remove`, so that instead of taking an element, it takes a predicate `p` and removes the first element of the list which satisfies the predicate (Listing 3). In order to do that and avoid similar pitfalls as in our first attempt to define `remove`, we need to ensure that the third disjunct succeeds only when the predicate `p` fails on the head element of `xs`. Thus we need a general *negation operator*. Unfortunately, none of the existing MiniKanren implementations provide this feature.

In the world of Prolog, negative reasoning is usually implemented using so-called *negation as failure* approach. Under this rule, a goal ¬p succeeds whenever p fails, and it fails whenever p succeeds. Unfortunately, negation as failure is unsound if the negated goal p contains free variables. For example, the query `run (q ≡ 0) ∧ ¬(q ≡ 1)` succeeds, although `run ¬(q ≡ 1) ∧ (q ≡ 0)` fails. The failure in the last case is due to the fact that at the time of negation execution the variable `q` is free, thus `q ≡ 1` succeeds, and conversely ¬(q ≡ 1) fails. This example demonstrates that negation as failure goes against the philosophy of MiniKanren, which positions itself as a pure relational language without non-logical features.

In this work we present an implementation of the negation operator based on the method of *constructive negation*. It overcomes several shortcomings of negation as failure and provides a more expressive form of negative reasoning than disequality constraints. The idea of this method is to constructively build a stream of answers for the negated goal. It is achieved by first collecting all answers to the positive version of the goal and then negating them. The constructive negation approach while being an improvement over negation as failure still has its own drawbacks. For example, applying the negation operator to the goal that has infinitely many answers results in a non-terminating computation.

Although constructive negation was first proposed as an extension of Prolog, to the best of our knowledge our work is the first attempt to adapt it for MiniKanren. We have developed a prototype implementation for OCanren [Kosarev and Boulytchev 2018], a MiniKanren dialect embedded in OCaml. Like the rest of

MiniKanren, our version of constructive negation is developed in a pure functional style using persistent data structures. This makes it different from the earlier implementations for Prolog.

Our paper is structured as follows. In Section 2 we give more examples of constructive negation usage. Section 3 describes our implementation. Next, Section 4 presents the evaluation of the negation on a series of examples. In Section 5 we discuss the limitations of our approach and directions for future work. Section 6 reviews related works. The final Section 7 concludes.

2 MOTIVATING EXAMPLES

In this section we give further motivation for adding the negation into relational programming. We present several examples of how the negation can be used.

2.1 Relational If-then-else

In the Prolog one can simulate the conditional if-then-else operator using so-called *soft cut* [Naish 1995]. The behavior of the soft cut `c →* t ; e` can be described as follows:

- if the goal `c` succeeds (i.e. produces at least one answer) then the result of `c →* t ; e` is equivalent to `c ∧ t`;
- if the goal `c` fails (i.e. produces no answers at all) then the result of `c →* t ; e` is equivalent to `e`.

The soft cut is an example of a non-relational feature. Such features usually do not compose well in the sense that they are sensitive to the order in which they appear in a program. For example, consider the goal `(c →* t ; e) ∧ g`, and assume that `c ∧ g` always fails regardless of the order of conjuncts. Then if `c` succeeds the result of the above goal will be equivalent to `c ∧ t ∧ g` and thus it will fail. Suppose we reorder the conjuncts as follows: `g ∧ (c →* t ; e)`. Now the goal `c`, when executed after `g`, certainly fails, and thus the result of the whole goal will be equivalent to `g ∧ e` which does not fails necessarily. One can see that the simple reordering of subgoals in the program can lead to the different results.

With the help of constructive negation if-then-else can be simply expressed as follows:

```
let ifte c t e =
  (c ∧ t) ∨ (¬ c ∧ e)
```

The behavior of if-then-else defined in this way subsumes the behavior of the soft cut. That is, every answer to the query `run (c →* t ; e)` is also an answer to the query `run (ifte c t e)`. Moreover, `ifte` is not sensitive to the order of subgoals in the program.

2.2 Classical Implication and Universal Quantification

With the negation added to the language, one can easily express other connectives of the classical first-order logic, namely the implication and universal quantification[1], using the well-known equivalences.

```
let (⇒) : goal → goal → goal =
  fun g1 g2 → ¬ g1 ∨ (g1 ∧ g2)

let forall : ('a → goal) → goal =
  fun g → ¬ fresh (x) (¬ g x)
```

However, we should make a few remarks here. It is well known that the search implemented in conventional MiniKanren is complete, meaning that every answer to an arbitrary query will be found eventually. In the presence of constructive negation (and thus implication and universal quantification defined through negation)

[1] It is easy to see that `c ⇒ t` is equivalent to `ifte c t succ`

the search becomes incomplete as we will later see. Moreover, constructive negation is computationally heavy and thus the double usage of it, as in the definition of `forall`, can be inefficient in some cases.

Despite all this trouble we have found that the above definitions are still useful. Some of the previous MiniKanren implementations introduced *eigen* variables, adopted from λProlog [Miller and Nadathur 2012]. Eigen variables act as a universally quantified variables. Yet, to the best of our knowledge, there is no sound implementation of eigen variables with the support of disequality constraints. We observed that our implementation of universal quantifications through double negation works nicely with disequalities (we give some examples in the Section 4).

2.3 Graph Unreachability Problem

One of the classical examples of negation application in logic programming is a problem of checking whether one node of the graph is unreachable from the another [Przymusinski 1989]. The code on Listing 4 defines binary relation `edge`, which binds pairs of nodes in graph, connected by some edge, and binary relation `reachable`, which is nothing more than a transitive closure of the `edge` relation. Then the relation `unreachable` is simply negation of `reachable`.

```
let edge x y =
  (x, y) ≡ ('a', 'b') ∨
  (x, y) ≡ ('b', 'a') ∨
  (x, y) ≡ ('b', 'c') ∨
  (x, y) ≡ ('c', 'd')

let reachable x y =
  x ≡ y ∨
  fresh (z) (
    edge x z ∧ reachable z y
  )

let unreachable x y =
  ¬(reachable x y)
```

Listing 4. Unreachability in a graph

Given this definition the query `run unreachable 'c' 'a'` will succeed. A knowledgeable reader might notice that constructive negation is not necessary in this case because negation as failure will deliver the same result. But the query `run unreachable 'c' q` will fail under negation as failure because of the free variable `q` which will appear under the negation. However constructive negation will succeed delivering the constraint `q ≠ 'd'`.

2.4 Unreachability in Labeled Transition Systems

One can consider a special kind of graphs — *labeled transition systems* [Baier and Katoen 2008]. Labeled transition system is defined by a set of states S, a set of labels L and a ternary transition relation $R \subseteq S \times L \times S$. By existential quantification over labels one can then obtain a binary relation. Taking its transitive closure gives the reachability relation. The negation of the reachability relation can be used to check that some state s' is not reachable from the initial state s. The Listing 5 shows an encoding of an abstract labeled transition system in OCanren.

Labeled transition systems are often used to describe the behavior of imperative languages. Although the naive encoding of transition relation in OCanren with simple enumeration of reachable states is often not tractable for

checking reachability (or unreachability) in huge state spaces arising in practical imperative programs, it still can be used, for example, for the task of prototyping the semantics of such languages.

```
module type LTS = sig
  type state
  type label

  val transition : state → label → state → goal
end

module LTSExt (T : LTS) = struct

  let reachable : T.state → T.state → goal =
    fun s s'' →
      s ≡ s''
      ∨
      fresh (l s') (
        T.transition s l s' ∧
        reachable s' s''
      )

  let unreachable : T.state → T.state → goal =
    fun s s' →
      ¬(reachable s s')
end
```

Listing 5. Unreachability in a labeled transition system

3 IMPLEMENTATION

In this section we present our implementation of constructive negation. We start with the general ideas behind the method (Section 3.1). We describe how the constructive negation behaves in concrete examples, starting from trivial ones and moving to more sophisticated. During this presentation, we will observe, that in order to implement constructive negation, we need a solver for universally quantified disequality constraints. We will show that such solver can be implemented on top of existing MiniKanren disequality solver with just a few modifications (Section 3.2). In the Section 3.3, we describe how the OCanren search can be extended to support constructive negation. We will also discuss how negation interacts with recursion and present the notion of *stratification* (Section 3.4).

3.1 General Ideas

Constructive negation is based on the following idea: given a goal ¬g, one can construct an answer for this goal by collecting all answers to its positive version g and then taking their complementation. In order to do that, a notion of "negation" of an answer is needed. Since each answer can be matched to some logical formula [Przymusinski 1989; Stuckey 1991], a "negation" of an answer corresponds to the logical negation of this formula.

Example 1. Consider the goal $\neg(\mathtt{q} \equiv 1 \ \wedge \ \mathtt{r} \equiv 2)$. Its positive version $(\mathtt{q} \equiv 1) \ \wedge \ (\mathtt{r} \equiv 2)$ has single answer, a substitution $\{\mathtt{q} \mapsto 1, \mathtt{r} \mapsto 2\}$ which corresponds to the formula $q = 1 \wedge r = 2$. By negating this formula we obtain $q \neq 1 \vee r \neq 2$. This formula still can be represent by a single substitution $\{\mathtt{q} \mapsto 1, \mathtt{r} \mapsto 2\}$. However, we now treat this substitution differently, as a *disequality constraint*.

Some MiniKanren implementations (including OCanren) already have the support for disequality constraints. A programmer can use them with the help of $\not\equiv$ primitive, as we have seen in the `remove` example (Listing 2). Usually, the support for disequalities is implemented as follows.

- A current state is maintained during the search. The state consists of a substitution, which represents positive information, and a disequality constraint store, which represents negative information. A constraint store can be implemented simply as a list of substitutions, or as a more efficient data structure.
- Each time a subgoal of the form $\mathtt{t} \not\equiv \mathtt{u}$ is encountered during the search, its satisfiability in current substitution is checked. If it is satisfiable, then disequality is added to the constraint store.
- Each time a subgoal of the form $\mathtt{t} \equiv \mathtt{u}$ is encountered during the search, the current substitution is refined by the result of unification of terms `t` and `u`. Then the satisfiability of disequality constraints is rechecked in the refined substitution.

Various optimizations can be applied to the scheme above. For example, there is no need to recheck every disequality in the store after each unification. We will not discuss these optimizations here, as they are irrelevant to our goals.

Unfortunately, as the next example illustrates, disequalities presented above are not sufficient to implement constructive negation.

Example 2. Consider a goal $\neg(\textbf{fresh} \ (\mathtt{x}) \ (\mathtt{q} \equiv (\mathtt{x}, \ \mathtt{x}))$, which states that `q` should not be equal to some pair of identical terms. The subgoal **fresh** $(\mathtt{x})$ $(\mathtt{q} \equiv (\mathtt{x}, \ \mathtt{x}))$ succeeds, delivering the substitution $\{\mathtt{q} \mapsto (\mathtt{x}, \mathtt{x})\}$. Because the variable `x` occurs under **fresh**, the corresponding formula is existentially quantified: $\exists x.\, q = (x, x)$. By the negation of this formula we obtain $\forall x.\, q \neq (x, x)$. This formula differs from disequality formula from example 1 as it contains universally quantified variable `x`.

Thereby, in order to support the negation of goals, containing **fresh**, we need to extend disequality constraint solver, so it can check the satisfiability of *universally quantified disequality constraints* in the form $\forall \overline{x}.\, t \neq u$ [2] [Chan 1988; Stuckey 1991]. Later, in Section 3.2 we will show how it can be done, for now let us assume we have such a solver.

We took care about **fresh** under negation. It led us to a more complicated representation of the state. During the search we maintain a pair of a current substitution and a universally quantified disequality constraint store. But now an interesting question arises: is this representation closed under negation? If we perform negation one more time, will we obtain a finite number of states in a similar form?

Lucky for us, it is the case. In order to verify it, let us consider a logical formula which corresponds to the representation of the state:

$$\exists \overline{x}. \left(\bigwedge_i (v_i = t_i) \wedge \bigwedge_j \forall \overline{y_j}. \bigvee_k (w_{jk} \neq u_{jk}) \right) \tag{1}$$

Here v_i and w_{jk} denote some variables, t_i and u_{jk} denote some terms. Existentially quantified variables $\overline{x}$ correspond to the variables occurred under **fresh**. The left conjunct corresponds to the substitution, the right conjunct corresponds to the constraint store. The constraint store itself is represented as a conjunction of individual universally quantified disequalities. As we have seen in example 1, each disequality corresponds to a

[2] $\overline{x}$ notation denotes a vector of variables, t and u are terms that may or may not contain variables from $\overline{x}$

disjunction of individual inequalities over variables[3]. Besides existential variables $\overline{x}$ and universal variables $\overline{y_j}$, there are free variables $\overline{q}$ that may occur in the formula (such as variables q and r in the examples 1, 2).

By logical negation of the above formula, we get the following:

$$\forall \overline{x}. \left(\bigvee_i (v_i \neq t_i) \vee \bigvee_j \exists \overline{y_j}. \bigwedge_k (w_{jk} = u_{jk}) \right) \tag{2}$$

The left disjunct, which corresponded to the substitution in the original formula, now became a disequality constraint. Looking at the right disjunct, we can see that each disequality has transformed into a substitution. However, there is one subtlety here. Variables w_{jk} may be universally quantified, that is $w_{jk} \in \overline{x}$ for some j, k. Moreover, the terms u_{jk} may contain universally quantified variables as well, $Vars(u_{jk}) \subseteq \overline{x}$ for some j, k. In the section 3.2 we will show, that each disjunct $\exists \overline{y_j}. \bigwedge_k (w_{jk} = u_{jk})$ is either unsatisfiable or could be rewritten as $\exists \overline{y_j}. \bigwedge_{k^*} (w^*_{jk^*} = u^*_{jk^*})$, such that universally quantified variables do not occur among variables $w^*_{jk^*}$ or in terms $u^*_{jk^*}$ [Liu et al. 1999].

Taking this into account, we can rewrite the formula 2 as follow:

$$(\bigvee_j \exists \overline{y_j}. \bigwedge_{k^*} (w^*_{jk^*} = u^*_{jk^*})) \vee \forall \overline{x}. \bigvee_i (v_i \neq t_i) \tag{3}$$

In the obtained formula each disjunct corresponds to one state in the form, similar to the given in formula 1. In the left disjunct, each sub-disjunct corresponds to a substitution with an empty disequality constraint, the right disjunct corresponds to the single universally quantified disequality constraint with an empty substitution. Thereby, the proposed representation of states is closed under negation.

One can perform further manipulations on the formula 3. Given that equivalence $a \vee \neg b = a \wedge b \vee \neg b$ holds in classical logic, we can rewrite formula 3 in the following way:

$$(\exists \overline{x}. \bigwedge_i (v_i = t_i) \wedge \bigvee_j \exists \overline{y_j}. \bigwedge_{k^*} (w^*_{jk^*} = u^*_{jk^*})) \vee \forall \overline{x}. \bigvee_i (v_i \neq t_i) \tag{4}$$

The latter transformation, while vacuous from the logical point of view, could improve the performance of the search in practice. It follows from the fact, that the subpart $\exists \overline{x}. \bigwedge_i (v_i = t_i)$ of the formula extends each substitution with additional mappings, thus delivering more positive information. If the negation constitutes a subpart of some larger goal, this positive information could lead to the earlier failure during the search.

Finally let us consider the negation in general case. A goal can be matched to a logical formula in the following way. Each answer to the goal corresponds to the state which itself corresponds to a logical formula in the form similar to one in formula 1. Goal can have multiple answers (even an infinite number). In the corresponding logical formula these answers will be connected by the disjunction:

$$\bigvee_n \left(\exists \overline{x_n}. \left(\bigwedge_i (v_{ni} = t_{ni}) \wedge \bigwedge_j \forall \overline{y_{nj}}. \bigvee_k (w_{njk} \neq u_{njk}) \right) \right) \tag{5}$$

The negation of this formula after an application of the transformations described above will become:

$$\bigwedge_n \left((\exists \overline{x_n}. \bigwedge_i (v_{ni} = t_{ni}) \wedge \bigvee_j \exists \overline{y_{nj}}. \bigwedge_{k^*} (w^*_{njk^*} = u^*_{njk^*})) \vee \forall \overline{x_n}. \bigvee_i (v_{ni} \neq t_{ni}) \right) \tag{6}$$

[3] We will give further explanation in section 3.2

In this way, we have obtained the result of the negation of the goal as a conjunction of the negation of individual answers to the positive version of the goal. However, one of the pitfalls of this construction is that the process will not terminate if the positive version of the goal has an infinite number of answers. Thus, in the general case, it makes the OCanren search incomplete for the goals involving negation.

3.2 Constraint Solver, Formally

In this section we will formally define the satisfiability of universally quantified disequality constraints and quantified equalities, mentioned in section 3.1. We will also present a simple decision procedure for satisfiability checking. To do that we need a rather standard notions of terms, substitutions, unifiers, etc. For the sake of completeness we give these definitions here. We also mention a standard unification algorithm as a decision procedure for equality constraints. This view of unification bridges the gap between the conventional and constraint logic programming.

Let us start with definitions of terms and substitutions.

Definition 1. Given an infinite set of variables V and a finite set of constructor symbols $C = \{C^i_{n_i}\}_i$, each with associated arity n_i, the set of *terms* T is inductively defined as follow:

- $\forall v \in V.\, v \in T$ — every variable is a term;
- $\forall c_k \in C.\, \forall t_1 \ldots t_k \in T.\, c_k(t_1, \ldots, t_k) \in T$ — every application of k-ary constructor symbol to k terms is a term.

From now on we will assume that terms are untyped (unsorted) and that there exists an infinitely many constructors of any arity.

Definition 2. Two terms t and u are *syntactically equal*, denoted as $t = u$, iff either

- $t = v$ and $s = v$ for some variable v;
- $t = c_k(t_1, \ldots, t_k)$, $s = c_k(s_1, \ldots, s_k)$ and $\forall i \in \{1..k\}.\, t_i = s_i$.

Definition 3. A substitution σ is a function from variables to terms: $\sigma : V \to T$, s.t. $\sigma(x) \neq x$ only for a finite number of variables. Every substitution σ can be represented as a finite list of pairs $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. By $dom(\sigma)$ we denote the set $\{x_1, \ldots, x_n\}$ and by $codom(\sigma)$ we denote the set $\{t_1, \ldots, t_n\}$. We denote empty substitution as $\top$. We also extend the set of substitutions defined above with the one additional element $\bot$.

Definition 4. A substitution can be *applied to a term*. The result of an application of σ ($\sigma \neq \bot$) to t, written as $t\sigma$, is a term defined in the following way:

- $x\sigma \triangleq \sigma(x)$;
- $c_k(t_1, \ldots, t_k)\sigma \triangleq c_k(t_1\sigma, \ldots, t_k\sigma)$.

The result of the application of $\bot$ to any term is undefined.

Lemma 1. Given two substitutions σ and θ, if $\forall v \in V.\, \sigma(v) = \theta(v)$ then $\forall t.\, t\sigma = u\theta$.

Proof. Can be proved by the induction on t. □

We are now ready to define the the satisfiability of *equality constraint*.

Definition 5. Equality constraint $t \equiv u$ is

- *satisfiable* if $\exists \sigma.\, t\sigma = u\sigma$; such σ is called a *unifier* of t and u;
- *unsatisfiable* otherwise.

Next, we show that the standard unification algorithm can be seen as a decision procedure for checking satisfiability of equality constrains. Before that we need to introduce several definitions.

Definition 6. A term t is *subsumed* by the term u, denoted as $t \sqsubseteq u$, iff $\exists\sigma.\, t = u\sigma$. If $t \sqsubseteq u$ we will also say that t *is a more specific* term than u, or u *is a more general* term than t.

Definition 7. A substitution σ is *subsumed* by a substitution τ, denoted as $\sigma \sqsubseteq \tau$, iff $\forall t.\, t\sigma \sqsubseteq t\tau$. If $\sigma \sqsubseteq \tau$ we also say that σ *is a more specific substitution* than τ, or σ *is a more general substitution* than τ.

Definition 8. Given terms t and u their unifier σ is called the *most general unifier*, iff for every other unifier τ σ is more general that τ, $\tau \sqsubseteq \sigma$.

Theorem 1. Given terms t and u they are either not unifiable (meaning that $\forall\sigma.\, t\sigma \neq u\sigma$), or there exists their most general unifier.

Proof. Proof of this statement can be found in [Robinson et al. 1965]. Proof of the termination and correctness of the unification algorithm, used by the most MiniKanren implementations, can be found in [Kumar and Norrish 2010]. □

From now on we will denote the most general unifier of two terms t and u as $mgu(t, u)$. In case of t is not unifiable with u, we assume that $mgu(t, u) = \bot$.

Remark 1. Note we can associate with an equality constraint $t \equiv u$ a logical first-order formula $t = u$. Additionally, we can associate with each substitution a logical first-order formula by the following rules:

- empty substitution $\top$ is associated with the truth constant $\top$;
- $\bot$ is associated with the falsity constant $\bot$;
- $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is associated with the formula $x_1 = t_1 \wedge \cdots \wedge x_n = t_n$.

Now we can have yet another view on unification. We can say that giving the problem of deciding satisfiability of the formula $t = u$, a unification algorithm reduces it to checking satisfiability of a simpler formula, which corresponds to a substitution. Such a formula is either trivially unsatisfiable (in the case of $\bot$) or trivially satisfiable.

Later on we will need the notion of idempotent substitution and idempotent unifier.

Definition 9. Substitution σ is *idempotent* iff $\forall t.\, t\sigma\sigma = t\sigma$

Lemma 2. If two terms are unifiable, there exists their idempotent unifier.

Proof. For the proof of this statement (for the case of unification algorithm, used in MiniKanren), we refer an interested reader to [Kumar and Norrish 2010]. □

We are ready to move on to disequality constraints. We start with the regular (not quantified) disequalities.

Definition 10. A disequality constraint $t \not\equiv u$ is

- *satisfiable* if $\exists\sigma.\, t\sigma \neq u\sigma$;
- *unsatisfiable* otherwise.

Next lemma gives us a simple decision procedure for checking satisfiability of disequalities.

Lemma 3. Disequality constraint $t \not\equiv u$ is

- *satisfiable* if $mgu(t, u) \neq \top$;
- *unsatisfiable* otherwise.

Proof. Let $\theta = mgu(t, u)$. Let us first show that if $\theta = \top$ then disequality is unsatisfiable. By the definition of $\top$ we have $t\theta = t$ and $u\theta = u$, by the definition of unifier $t\theta = u\theta$, and thus $t = u$. From that, it is easy to show that $\forall\sigma.\, t\sigma = s\sigma$, which means that the disequality is unsatisfiable according to the definition 10. If $\theta \neq \top$ then there exists a substitution σ, s.t. $\theta \sqsubseteq \sigma$ (e.g. $\sigma = \top$). Since θ is most general unifier, and σ is more general that θ, then σ is not a unifier, and thus $t\sigma \neq u\sigma$. □

Given lemma 3, the satisfiability of constraint $t \not\equiv u$ can be checked easily. One need to compute $mgu(t, u)$ and if it is not an empty substitution, then constraint is satisfiable.

Remark 2. Given disequality constraint $t \not\equiv u$, a substitution $mgu(t, u)$, can be matched to the logical formula in the following way:

- the empty substitution $\top$ is associated with falsity constant $\bot$;
- $\bot$ is associated with the truth constant $\top$;
- $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is associated with the formula $x_1 \neq t_1 \vee \cdots \vee x_n \neq t_n$.

The following definition introduces universally quantified disequalities.

Definition 11. Universally quantified disequality constraint $\forall \overline{x}.\, t \not\equiv u$ is

- *satisfiable* iff $\exists \sigma.\, \forall \tau,\, dom(\tau) \subseteq \overline{x}.\, t\tau\sigma \neq u\tau\sigma$
- *unsatisfiable* iff $\forall \sigma.\, \exists \tau,\, dom(\tau) \subseteq \overline{x}.\, t\tau\sigma = u\tau\sigma$

For the decision procedure of this type of disequalities, we need one auxiliary lemma.

Lemma 4. Given terms t and u consider $\theta = mgu(t, u)$. Assume without the loss of generality that $\overline{x} \not\subseteq codom(\theta)$ (if $v \mapsto x \in \theta$ for some $x \in \overline{x}$ consider $\hat{\theta}$ such that it is equal to θ except that instead of mapping v to x it maps x to v). Universally quantified disequality constraint $\forall \overline{x}.\, t \not\equiv u$ is

- *satisfiable* if $\theta = \bot$ or $dom(\theta) \not\subseteq \overline{x}$
- *unsatisfiable* if $dom(\theta) \subseteq \overline{x}$

Proof. First, let us show that $\theta = \bot$ or $dom(\theta) \not\subseteq \overline{x}$ implies satisfiability. We need to show that $\exists \sigma.\, \forall \tau,\, dom(\tau) \subseteq \overline{x}.\, t\tau\sigma \neq u\tau\sigma$. Take $\sigma = \top$. Thus $t\tau\sigma = t\tau$ and $u\tau\sigma = u\tau$. It is left to show that $\forall \tau,\, dom(\tau) \subseteq \overline{x}.\, t\tau \neq u\tau$. If $\theta = \bot$ then t and u are not unifiable, which implies the above statement. Otherwise, consider some τ such that $dom(\tau) \subseteq \overline{x}$. If $t\tau = u\tau$ then τ is a unifier of t and u. Thus $\tau \sqsubseteq \theta$. If we will show that $dom(\theta) \subseteq dom(\tau) \subseteq \overline{x}$ we will get a contradiction with the our assumptions and therefore $t\tau \neq u\tau$. Indeed, consider $v \in dom(\theta)$. If $v \notin dom(\tau)$ consider two cases:

- $v \mapsto w \in \theta$ for some $w \in V$. From the assumptions follows that $w \notin \overline{x}$ and thus $w \notin dom(\tau)$. Consider the term $f(v, w)$ for some binary constructor f. It is easy to see that $f(v, w)\tau = f(v, w) \not\sqsubseteq f(w, w) = f(v, w)\theta$, which contradicts $\tau \sqsubseteq \theta$. Thus it should be that $v \in dom(\tau)$.
- $v \mapsto s \in \theta$ for some term $s \notin V$. Then, trivially $v\tau = v \not\sqsubseteq s = v\theta$, which contradicts $\tau \sqsubseteq \theta$. Thus it should be that $v \in dom(\tau)$.

Finally, let us show that if $dom(\theta) \subseteq \overline{x}$ then $\forall \sigma.\, \exists \tau,\, dom(\tau) \subseteq \overline{x}.\, t\tau\sigma = u\tau\sigma$. Indeed, consider some σ. Take $\tau = \theta$. Then $t\tau = u\tau$ and thus $t\tau\sigma = u\tau\sigma$. □

By the above lemma, if the substitution $mgu(t, u)$ maps only universally quantified variables, then the disequality is unsatisfiable and satisfiable otherwise.

Finally, it is left to show how to check satisfiability of the quantified equalities of the form $\forall \overline{x}.\, \exists \overline{y}.\, t \equiv u$. As we will see soon, if the constraint of this form is satisfiable, then the logical formula, corresponding to the constraint, $\forall \overline{x}.\, \exists \overline{y}.\, t = u$ is equivalent to the formula $\exists \overline{y^*}.\, \bigwedge_i v_i = t_i$ such that $v_i \in V$ and $v_i \notin \overline{x}$ and $Vars(t_i) \cap \overline{x} = \emptyset$ for all i [Liu et al. 1999].

Definition 12. Quantified equality constraint of the form $\forall \overline{x} \exists \overline{y}.\, t \equiv u$ is

- *satisfiable* iff $\exists \sigma.\, \forall \tau,\, dom(\tau) \subseteq \overline{x}.\, \exists \phi,\, dom(\phi) \subseteq \overline{y}.\, t\phi\tau\sigma = u\phi\tau\sigma$
- *unsatisfiable* iff $\forall \sigma.\, \exists \tau,\, dom(\tau) \subseteq \overline{x}.\, \forall \phi,\, dom(\phi) \subseteq \overline{y}.\, t\phi\tau\sigma \neq u\phi\tau\sigma$

Lemma 5. If terms t and u are not unifiable then the constraint is unsatisfiable. Otherwise let θ be an idempotent unifier of t and u (Lemma 2 states that if terms are unifiable there exists their idempotent unifier). Let $\theta_y \triangleq \{y \mapsto t \mid y \mapsto t \in \theta \wedge y \in \overline{y}\}$. Let $\hat{\theta} = \{v \mapsto t \mid v \mapsto t \in \theta \wedge v \notin \overline{y}\}$. Then the quantified equality constraint of the form $\forall \overline{x} \exists \overline{y}.\, t \equiv u$ is

- *satisfiable* if $dom(\hat{\theta}) \cap \overline{x} = \emptyset$ and $\forall p \in codom(\hat{\theta}).\, Vars(p) \cap \overline{x} = \emptyset$
- *unsatisfiable* if $dom(\hat{\theta}) \cap \overline{x} \neq \emptyset$ or $\exists p \in codom(\hat{\theta}).\, Vars(p) \cap \overline{x} \neq \emptyset$

Proof. First, it is obvious that if t and u are not unifiable then the constraint is unsatisfiable. Next, let us prove the statement involving satisfiability. We need to show that if the given condition is met, then $\exists \sigma.\, \forall \tau, dom(\tau) \subseteq \overline{x}.\, \exists \phi, dom(\phi) \subseteq \overline{y}.\, t\phi\tau\sigma = u\phi\tau\sigma$. Take $\sigma = \hat{\theta}$. Given an arbitrary τ such that $dom(\tau) \subseteq \overline{x}$ take ϕ to be equal to θ_y. To complete the proof we will need an auxiliary statement.

- $\forall s.\, s\tau\hat{\theta} = s\hat{\theta}\hat{\tau}$ where $\hat{\tau} \triangleq \{x \mapsto t\hat{\theta} \mid x \mapsto t \in \tau\}$.

  Proof. By the Lemma 1 it is sufficient to show that $\forall v \in V.\, v\tau\hat{\theta} = v\hat{\theta}\hat{\tau}$. Consider the cases:
  – $v \in \overline{x}$. Then $v\tau\hat{\theta} = \tau(v)\hat{\theta}$. Since $dom(\hat{\theta}) \cap \overline{x} = \emptyset$, $\hat{\theta}(v) = v$. Then $v\hat{\theta}\hat{\tau} = v\hat{\tau}$ and by the construction $v\hat{\tau} = \tau(v)\hat{\theta}$.
  – $v \notin \overline{x}$. Then $v\tau\hat{\theta} = \hat{\theta}(v)$. Since $\forall p \in codom(\hat{\theta}).\, Vars(p) \cap \overline{x} = \emptyset$, $v\hat{\theta}\hat{\tau} = v\hat{\theta}$ and trivially $v\hat{\theta} = \hat{\theta}(v)$.

  ■

By this statement $t\theta_y\tau\hat{\theta} = t\theta_y\hat{\theta}\hat{\tau}$ and $u\theta_y\tau\hat{\theta} = u\theta_y\hat{\theta}\hat{\tau}$. Because θ is idempotent $t\theta = t\theta_y\hat{\theta}$ and $u\theta = u\theta_y\hat{\theta}$. Finally, since θ is a unifier $t\theta = u\theta$.

It is left to prove the statement involving unsatisfiability. In fact, we will prove more general statement.

- Let $\tilde{t}$ and $\tilde{u}$ be two arbitrary unifiable terms, let $\tilde{\theta}$ be their unifier. Then
  $\forall \overline{x} \subseteq dom(\tilde{\theta}) \cup \bigcup_{p \in codom(\tilde{\theta})} Vars(p), \overline{x} \neq \emptyset.\, \forall \sigma.\, \exists \tau, dom(\tau) \subseteq \overline{x}.\, \tilde{t}\tau\sigma \neq \tilde{u}\tau\sigma$

  Proof. By the induction on t:
  – $\tilde{t} = v$ for some $v \in V$. Consider the cases for u:
    * $\tilde{u} = w$ for some $w \in V$. Then $\tilde{\theta} = \{v \mapsto w\}$ and either $v \in \overline{x}$ or $w \in \overline{x}$.
      Let the former be true (the other case is similar). Given some arbitrary σ
      take $\tau \triangleq \{v \mapsto z \mid z \in V \setminus (dom(\sigma) \cup \bigcup_{p \in codom(\sigma)} Vars(p))\}$. Then $v\tau\sigma = z$ and $w\tau\sigma \neq z$.
    * $\tilde{u} = g(\tilde{u_1}, \ldots, \tilde{u_m})$ for some constructor g. Then $\tilde{\theta} = \{v \mapsto u\}$. If $v \in \overline{x}$ then pick some constructor $f \neq g$ (because we assume there exists an infinite number of constructor symbols, we can always do it). Take $\tau \triangleq \{v \mapsto f(z_1, \ldots, z_n) \mid z_i \in V\}$. Then $v\tau\sigma = f(\tilde{t_1'}, \ldots, \tilde{t_n'})$ and $\tilde{u}\tau\sigma = g(\tilde{u_1'}, \ldots, \tilde{u_m'})$, and thus these terms are not equal. If $v \notin \overline{x}$ then take some $x \in \overline{x}$. For some i it should be that $x \in Vars(\tilde{u_i})$. Given σ consider $\sigma(v)$, pick some s such that $\sigma(v) \neq g(\tilde{u_1}, \ldots, \tilde{u_m})\{x \mapsto s\}$ (it can be done by the induction on $\sigma(v)$). Then $\tau \triangleq \{x \mapsto s\}$.
  – $\tilde{t} = f(\tilde{t_1}, \ldots, \tilde{t_n})$. Consider the cases for u:
    * $\tilde{u} = w$ for some $w \in V$. Then the proof proceeds in the same way as in the previous case.
    * $\tilde{u} = f(\tilde{u_1}, \ldots, \tilde{u_n})$ ($\tilde{u}$ cannot be equal to some constructor $g \neq f$ by our assumption of unifiability of terms). Then by our assumption $\tilde{t_1} \equiv \tilde{u_1} \wedge \cdots \wedge \tilde{t_n} \equiv \tilde{u_n}$. For some i it should be the case that $\overline{x} \subseteq dom(\tilde{\theta_i}) \cup \bigcup_{p \in codom(\tilde{\theta_i})} Vars(p)$ where $\theta_i \triangleq mgu(\tilde{t_i}, \tilde{u_i})$. By the induction for an arbitrary σ there exists τ such that $\tilde{t_i}\tau\sigma \neq \tilde{u_i}\tau\sigma$ and thus $\tilde{t}\tau\sigma \neq \tilde{u}\tau\sigma$.

  ■

□

Given this lemma, we compute $mgu(t, u)$ in order to check the constraint $\forall\overline{x}\exists\overline{y}.\ t \equiv u$. By $mgu(t, u)$ we can construct an idempotent substitution θ that is also a unifier of t and u. We take $\hat{\theta}$ — a part of θ that does not bind existentially quantified variables $\overline{y}$. Then we check if $\hat{\theta}$ binds variables from $\overline{x}$, or some term from codomain of $\hat{\theta}$ contains variables from $\overline{x}$. If it does then the constraint is unsatisfiable, because in this case we can always pick an assigment for $\overline{x}$ that will make the terms not unifiable. Otherwise it is satisfiable.

3.3 Extending the Search

In this section we describe how OCanren search interacts with negation. Also we finally present the code of negation operator itself.

In OCanren (as in any typical MiniKanren implementation) the search is implemented on top of backtracking lazy stream monad [Kiselyov et al. 2005]. During the search the current state is maintained. The state contains accumulated constraints plus some supplementary information stored in the environment (for example, the identifier of last allocated variable). A goal is simply a function which takes a state and returns a lazy stream of states. All logical primitives, such as individual constraints, conjunction, disjunction, and fresh variable introduction, can be implemented based on this representation of goals (an interested reader may refer to [Hemann and Friedman [n. d.]; Kiselyov et al. 2005]).

Now we can define the negation operator ¬ (see Listing 6). Let us describe it in details.

```
1  let (¬) g st =
2    let sts' = g st in
3    let cexs = Stream.map (diff st) sts' in
4    let sub ss cex =
5      let ss' = negate cex in
6      Stream.bind ss  (fun s  →
7      Stream.bind ss' (fun s' →
8        Stream.unit (merge s s')
9      ))
10   in
11   Stream.fold sub (Stream.unit st) cexs
```

Listing 6. Implementation of the negation operator

Negation operator is a function which takes a goal and returns a negated goal. Because a goal is itself a function taking a state, (¬) takes two arguments: the goal `g` and the state `st` (line 1).

The first step of constructive negation is to run the positive version of the goal, as code in line 2 does. We run it in the current state `st` and thus the call `g st` returns a stream of refined states `sts'`. Each state from this stream will contain the constraints from the original state `st` as its subpart. However, we need to negate only constraints originated from `g` solely. Thus, on the line 3 we map every state from the stream `sts'` to its *difference* with respect to the original substitution `st`. In order to compute difference of two states `st` and `st'` (Listing 7), given that `st` is more general that `st'` we need to compute difference of their substitutions and disequality constraints stores. The difference of substitution `s'` with respect to `s` (Listing 8) is just a substitution containing all mappings from `s'` which are not simultaneously in `s`. The difference of constraint store `c'` with respect to `c` is a constraint store

containing all disequalities that are in `c'` but not in `c` (Listing 8). As long as persistent data structures are used to implement substitutions and constraint stores, the `diff` can be computed[4].

Line 4 defines auxiliary function `sub` which takes two arguments: a stream of states `ss` and some state `cex`, and returns another stream. The purpose of this function it to "subtract" `cex` from every element of `ss`. It is done as follows. First, the state `cex` is negated as described in section 3.1 (line 5). As we have seen, as a result of the negation of a single state the stream of states (the disjunction of formulas) can be obtained. Thus the result of the call `negate cex` is the stream of states `ss'`. For every combination of some state `s` from the given stream `ss` (line 6) and some state `s'` from the stream `ss'` representing the result of negation (line 7) we compute their conjunction (line 8). The conjunction of two states is computed by the function `merge` (Listings 7, 8).

Finally, on the line 11 `fold` is called on the stream `cexs`, which is a stream of answers for the positive version of the goal `g`, with function `sub` defined above and the initial accumulator `Stream.unit st`. The function `Stream.fold` is implemented as a regular left fold over a possibly infinite list. Intuitively with folding over stream `cexs` we "subtract" from the original state `st` every answer obtained from the goal `g`.

[4] In the Listing 8 we present a simple representation of the constraint store as a list of substitutions. In the actual implementation, we use more sophisticated representation, that also provides the `diff` function. The simpler version presented here gives some intuition on how to implement `diff` for the constraint stores.

```
module State = struct
  type t = Env.t * Subst.t * CStore.t

  ...

  let diff (e, s, cs) (e', s', cs') =
    let e  = Env.diff e e' in
    let s  = Subst.diff s s' in
    let cs = CStore.diff cs cs' in
    (e, s, cs)

  let merge (e, s, cs) (e', s', cs') =
    let e = Env.merge e e' in
    match Subst.merge s s' with
    | None    → None
    | Some s  →
      let cs = CStore.merge cs cs' in
      match CStore.recheck s cs with
      | None    → None
      | Some cs → (e, s, cs)
end
```

Listing 7. Implementation of the auxiliary functions

```
module VarMap = Map.Make(Var)

module Subst = struct
  (* Substitution is a mapping from variables to terms *)
  type t = Term.t VarMap.t
  ...

  let diff s s' =
    VarMap.fold (fun v t a →
      if not (VarMap.mem v s) then
        VarMap.add v t a
      else a
    ) s' VarMap.empty

  let merge s s' =
    VarMap.fold (fun v t → function
      | None   → None
      | Some a → unify a v t
    ) s' (Some s)
end

module CStore = struct
  (* Constraint store is a list of substitutions *)
  type t = Subst.t list
  ...

  (* cs' must be obtained from cs by
   * the addition of new constraints
   *)
  let diff cs cs' =
    if cs' = cs then []
    else
      match cs' with
      | _ :: cs' → diff cs cs'
      (* cs' = [] implies cs = []  *)
      | _ → assert false

  let merge cs cs' =
    List.append cs' cs
end
```

Listing 8. Implementation of the auxiliary functions

3.4 Stratification

A negation, when combined with recursion, might become a source of confusion for a programmer. Consider the program in Listing 9. It encodes a two-players game. The positions in the game are given as single-character strings `'a'`, `'b'`, `'c'` and `'d'`. A binary relation `move` encodes the game field as the set of possible moves. An unary relation `winning` determines the set of winning positions, meaning that if the first player starts from some winning position, by making "good" moves the player has an opportunity to win the game. According to the definition of `winning`, the position is winning if there exists a move from this position to some non-winning position. Clearly, every position with no moves from it is losing.

Given the suchlike definition of `winning`, there is no doubt, that the position `'d'` is losing position, and thus the goal `winning 'd'` should fail, which, in turn, means that `winning 'c'` should succeed. However, whether the goal `winning 'a'` (or `winning 'b'`) should fail or succeed is not clear.

```
let move x y =
  (x, y) ≡ ('a', 'b') ∨
  (x, y) ≡ ('b', 'a') ∨
  (x, y) ≡ ('b', 'c') ∨
  (x, y) ≡ ('c', 'd')

let winning x =
  fresh (y) (
    (move x y) ∧ ¬(winning y)
  )
```

Listing 9. Encoding of two-players game

The problem with the semantics of program in Listing 9, originates from the interaction of negation and recursion. Definition of the relation `winning` refers to itself under negation. Logic programs that have this property are called *non-stratified* [Przymusinski 1989]. Vice versa, programs that do not have loops over negation, are called *stratified*.

Our current implementation handles only stratified programs. We leave the task of supporting non-stratified programs as a direction for future work.

4 EVALUATION

In this section, we present an evaluation of implemented constructive negation on a series of examples.

4.1 If-then-else

Using relational if-then-else operator, presented in section 2.1, we have implemented several higher-order relations over lists, namely `find` (Listing 10), `remove`[5] (Listing 11) and `filter` (Listing 12). These relations are almost identical (syntactically) to their functional implementations. We have tested that these relations can be run in various directions and produce the expected results. For example, the goal `filter p q q` with the predicate `p` equal to

```
fun l → fresh (x) (l ≡ [x])
```

stating that the given list should be a singleton list, starts to generate all singleton lists. Vice versa, the goal `filter p q []` with that same `p` generates all lists, constrained to be not a singleton list.

[5]Note, this implementation differs from the one in Section 1, but it is easy to see that these two are semantically equivalent.

Listings 13-16 give more concrete examples of queries to these relations. In the listing the syntax `run n q g` means running a goal `g` with the free variable `q` taking the first `n` answers ("$*$" denotes all answers). After the sign $\rightsquigarrow$ the result of the query is given. The result `fail` means that the query has failed. The result succ $\{\{a_1\}; \ldots \{a_n\}\}$ means that the query has succeeded delivering n answers. Each answer represents a set of constraint on free variables. Constraints are of two forms: equality constraints, e.g. $q = (1, \_.0)$, or disequality constraints, e.g. $q \neq (1, \_.0)$. The terms of the form $\_.i$ in the answer denote some universally quantified variables.

```
let find p e xs =
  fresh (x xs' ys') (
    xs ≡ x::xs' ∧
    ifte (p x)
      (e ≡ x)
      (find p e xs')
  )
```

Listing 10. A definition of `find` relation

```
let remove p xs ys =
  (xs ≡ [] ∧ ys ≡ [])
  ∨
  fresh (x xs' ys') (
    xs ≡ x::xs' ∧
    ifte (p x)
      (ys ≡ xs')
      (ys ≡ x::ys' ∧
       remove p xs' ys')
  )
```

Listing 11. A definition of `remove` relation

```
let filter p xs ys =
  (xs ≡ [] ∧ ys ≡ [])
  ∨
  fresh (x xs' ys') (
    xs ≡ x::xs' ∧
    (ifte (p x)
      (ys ≡ x :: ys')
      (ys ≡ ys')) ∧
    filter p xs' ys'
  )
```

Listing 12. A definition of `filter` relation

```
let p l = fresh (x) (l ≡ [x])
```

Listing 13. Definition of the predicate p

```
run 3 q (fresh (e) find p e q)
⇝ succ {
    { q = [_.0] :: _.1 }
    { q = _.0 :: [_.1] :: _.2;
        _.0 ≠ [_.3] }
    { q = _.0 :: _.1 :: [_.2] :: _.3;
        _.0 ≠ [_.4]; _.1 ≠ [_.5] }
  }
```

Listing 14. Example of queries to `find`

```
run * q (fresh (e) remove p q [[ ]])
⇝ succ {
    { q = [[_.0]; [ ]] }
    { q = [[ ]] }
    { q = [[ ]; [_.0]] }
  }

run 3 q (fresh (e) remove p q q)
⇝ succ {
    { q = [] }
    { q = [_.0], _.0 ≠ [_.1] }
    { q = [_.0; _.1];
        _.0 ≠ [_.2]; _.1 ≠ [_.3] }
  }
```

Listing 15. Example of queries to `remove`

```
run 3 q (filter p q q)
⇝ succ {
     { q = [ ] }
     { q = [_.0] }
     { q = [_.0; _.1] }
   }

run 3 q (filter p q [1])
⇝ succ {
     { q = [[1]] }
     { q = [_.0; [1]]; _.0 ≠ [_.1] }
     { q = [[1]; _.0]; _.0 ≠ [_.1] }
   }

run 3 q (filter p q [ ])
⇝ succ {
     { q = [] }
     { q = [_.0]; _.0 ≠ [_.1] }
     { q = [_.0; _.1];
            _.0 ≠ [_.2]; _.1 ≠ [_.3] }
   }
```

Listing 16. Example of queries to `filter`

4.2 Universal quantification

In the Section 2.2 we presented the `forall` goal constructor which is implemented through the double negation. We have observed, that although `forall g` does not terminate when the goal `g x` has an infinite number of answers (assuming `x` is a fresh variable), it does terminate in the case when `g x` has a finite number of answers. The behavior of `forall` in this case is sound even in the presence of disequality constraints or nested quantifiers.

The Table 1 gives some concrete examples. The left column contains the tested goals[6] and the right column gives the obtained results. For the results we use the same notation as in the previous section.

5 LIMITATIONS AND FUTURE WORK

In this section we discuss the limitations of constructive negation in general and our implementation in particular. Also we consider possible directions for future work.

5.1 Type Constraints

Although the program written in OCanren typechecks statically (thus, for example, preventing the user from unifying two terms of distinct types), at runtime the type information is erased. In the presence of even regular disequality constraints it can lead to the incorrect results. As an example, consider the following program:

[6] We typeset the goals in terms of first-order logic syntax instead of OCanren syntax for brevity and clarity.

$\forall x.\, x = q$	`fail`
$\forall x.\, \exists y.\, x = y$	`succ {[q = _.0]}`
$\forall x.\, \exists y.\, x = y \wedge y = q$	`fail`
$\forall x.\, q = (1, x)$	`fail`
$\forall x.\, \exists y.\, y = (1, x)$	`succ {[q = _.0]}`
$\forall x.\, \exists y.\, x = (1, y)$	`fail`
$\forall x.\, x \neq q$	`fail`
$\forall x.\, \exists y.\, x \neq y$	`succ {[q = _.0]}`
$\forall x.\, \exists y.\, x \neq y \wedge y = q$	`fail`
$\forall x.\, q \neq (1, x)$	`succ {[q ≠ (1, _.0)]}`
$(\exists x.\, q = (1, x)) \wedge (\forall x.\, q \neq (1, x))$	`fail`
$\forall x.\, (x, x) \neq (0, 1)$	`succ {[q = _.0]}`
$\forall x.\, (x, x) \neq (1, 1)$	`fail`
$\forall x.\, (x, x) \neq (q, 1)$	`succ {[q ≠ 1]}`
$\exists a\ b.\, q = (a, b) \wedge \forall x.\, (x, x) \neq (a, b)$	`succ {[q = (_.0, _.1); _.0 ≠ _.1]}`

Table 1. `forall` evaluation

```
type bool = true | false

let g =
  fresh (x y z : bool) (
    (x ≠ y)
    (y ≠ z)
    (z ≠ x)
  )
```

The goal g states that there exists at least three different non-equal terms of type `bool`, which, as we know, is not true. Yet the query `run g` will succeed.

In order to prevent unsoundness in cases like this, type information in the form of *type constraints* should be somehow attached to the variables at runtime. The satisfiability of type constraints then should be rechecked each time when the new disequality is added to some variable. An extension of OCanren with type constraints is a direction for future work.

5.2 Non-stratified Programs

As we have already discussed in the section 3.4 our current implementation handles only stratified logic programs. One of the possible extensions is to support non-stratified programs, such as one given in Listing 9, with respect to well-founded and/or stable model semantics (see section 6 for the details).

5.3 Negation of Goals With an Infinite Number of Answers

Consider the following program:

```
let zeros l =
  l ≡ [0]
  ∨
  fresh (l') (
    (l ≡ 0 :: l')
    (zeros l')
  )
```

The unary relation `zeros` defines lists consisting of zeros. Now, intuitively, the query `run ¬(zeros q)` should enumerate all lists that are not built out of zeros only. Yet this query will fail to deliver even a single answer. Why? Consider its operational behavior. First the positive version of the goal, that is `zeros q`, should be executed. Then all answers to this goal should be collected and complemented. However, there is an infinite number of answers to `zeros q` and thus this process will never terminate.

It is a significant drawback of constructive negation that the negation of the goal cannot be computed if the goal has an infinite number of answers. This limitation cannot be avoided in general, however in some cases it is possible to narrow the number of answers to some subgoal by the reordering of surrounding subgoals. For example, the query `run ¬(zeros q) ∧ (q ≡ [1])` can be executed in finite time by the reordering of conjuncts. It seems that the best strategy is to delay negative subgoals as long as possible, but we do not have a formal proof of that.

6 RELATED WORKS

There are two directions of work in the process of incorporating negative reasoning in the logic programming: the first considers the semantics of negation, and the second is focused mainly on implementation aspects.

The first attempt to give a semantics for negation in logic programming was done by Clark [Chan 1988; Clark 1978] with his completion semantics. It was then realized, that Clark's semantics has various drawbacks [Van Gelder et al. 1991].

Przymusinski [Przymusinski 1989] has studied the semantics of stratified logic programs. He introduced the notion of *perfect model semantics* for such programs. Stratified logic programs have a variety of good properties, including the property that each stratified program has a unique minimal model.

In an attempt to extend the semantics of negation to non-stratified programs the *well-founded semantics* was proposed [Van Gelder et al. 1991]. However, this semantics is three-valued, meaning that for some queries it can return answer `unknown`. For example, given the relation `winning` (section 3.4, listing 9), queries `winning 'a'` and `winning 'b'` would return `unknown`.

An alternative approach is *stable model semantics* [Gelfond and Lifschitz 1988]. Under this semantics, non-stratified logic program can have several stable models. Program, that defines `winning`, has two stable models, in one of these models goal `winning 'a'` succeeds and `winning 'b'` fails, in the other `winning 'a'` fails and `winning 'b'` succeeds. Logic programming under stable model semantics is also known under the name answer set programming (ASP).

The works [Dovier et al. 2000; Stuckey 1991] are theoretical studies of constructive negation in the context of constraint logic programming. They give a necessary and sufficient condition for the constraint structures that are compatible with constructive negation. Namely, the constraint structure should be *admissible closed*.

From an implementation side, Chen et al. [Chen et al. 1995; Chen and Warren 1996] developed a Prolog system based on SLG resolution, which is sound with respect to well-founded semantics. However, they used negation as failure with delaying of non-ground negative subgoals. [Liu et al. 1999] is an extension of this system with the support of the constructive negation. Works [Álvez et al. 2004; Barták 1998] implement a constraint logic programming systems with the support of constructive negation. Yet, as with our implementation, the constructive negation in these systems supports only equality and disequality constraints over first-order terms. We are not aware of any practical implementation that is parametric over arbitrary admissible closed constraint structures.

Many tools were developed to compute stable models of logic programs, among them are [Gebser et al. 2007; Giunchiglia et al. 2006]. These systems usually require to perform grounding of logic program. The problem of finding stable models of ground logic program then is encoded as propositional formula and solved by some SAT solver. Unfortunately, some logic programs do not have finite grounding, but even if a program has it, grounding may cause an exponential blow-up. Recently, a goal-directed system for computing stable models was developed [Arias et al. 2018; Marple et al. 2012, 2017]. To the best of our knowledge, it is the only ASP system, that does not require grounding. The key components of this system are the usage of tabling, constructive negation, coinductive logic programming, and non-monotonic reasoning check. It is an interesting and challenging task to extend MiniKanren with the support of stable model semantics in the spirit of this line of work.

7 CONCLUSION

We have presented an implementation of constructive negation for relational programming language OCanren, a dialect of MiniKanren. Unlike the negation as failure, constructive negation is consistent with the pure logical nature of MiniKanren. As we have demonstrated the negative reasoning increases the expressive power of relational language by allowing to compose more relations in a natural and intuitive form.

ACKNOWLEDGMENTS

We want to thank Dmitry Boulytchev, Ekaterina Verbitskaia and anonymous reviewers for valuable comments on a draft version of the paper. This work was partially supported by the grant 18-01-00380 from the Russian Foundation for Basic Research and the grant from JetBrains Research.

REFERENCES

Javier Álvez, Paqui Lucio, and Fernando Orejas. 2004. Constructive negation by bottom-up computation of literal answers. In *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, 1468–1475.

Joaquin Arias, Manuel Carro, Elmer Salazar, Kyle Marple, and Gopal Gupta. 2018. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming* 18, 3-4 (2018), 337–354.

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

Roman Barták. 1998. Constructive negation in clp (h). *submitted to CP* 98 (1998).

William E Byrd. 2010. Relational programming in miniKanren: techniques, applications, and implementations. (2010).

David Chan. 1988. Constructive negation based on the completed database. In *Proc. of ICLP-88*.

Weidong Chen, Terrance Swift, and David S Warren. 1995. Efficient top-down computation of queries under the well-founded semantics. *The Journal of logic programming* 24, 3 (1995), 161–199.

Weidong Chen and David S Warren. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM (JACM)* 43, 1 (1996), 20–74.

Keith L Clark. 1978. Negation as failure. In *Logic and data bases*. Springer, 293–322.

Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. 2000. A necessary condition for constructive negation in constraint logic programming. *Inform. Process. Lett.* 74, 3-4 (2000), 147–156.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer*. The MIT Press.

Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. 2007. clasp: A conflict-driven answer set solver. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 260–265.

Michael Gelfond and Vladimir Lifschitz. 1988. The stable model semantics for logic programming.. In *ICLP/SLP*, Vol. 88. 1070–1080.

Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning* 36, 4 (2006), 345.

Jason Hemann and Dan Friedman. [n. d.]. μkanren: A minimal functional core for relational programming, November 2013. *URL http://www.schemeworkshop.org/2013/papers/HemannMuKanren2013.pdf* ([n. d.]).

Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers:(functional pearl). *ACM SIGPLAN Notices* 40, 9 (2005), 192–203.

Dmitry Kosarev and Dmitry Boulytchev. 2018. Typed embedding of a relational language in OCaml. *arXiv preprint arXiv:1805.11006* (2018).

Ramana Kumar and Michael Norrish. 2010. (Nominal) unification by recursive descent with triangular substitutions. In *International Conference on Interactive Theorem Proving*. Springer, 51–66.

Julie Yuchih Liu, Leroy Adams, and Weidong Chen. 1999. Constructive negation under the well-founded semantics. *The Journal of Logic Programming* 38, 3 (1999), 295–330.

Kyle Marple, Ajay Bansal, Richard Min, and Gopal Gupta. 2012. Goal-directed execution of answer set programs. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*. ACM, 35–44.

Kyle Marple, Elmer Salazar, Zhuo Chen, and Gopal Gupta. 2017. The s (ASP) Predicate Answer Set Programming System. *The Association for Logic Programming Newsletter* (2017).

Dale Miller and Gopalan Nadathur. 2012. *Programming with higher-order logic*. Cambridge University Press.

Lee Naish. 1995. Pruning in logic programming. In *University of Melbourne*. Citeseer.

Teodor C Przymusinski. 1989. *On constructive negation in logic programming*. MIT Press Cambridge, Massachusetts.

John Alan Robinson et al. 1965. A machine-oriented logic based on the resolution principle. *J. ACM* 12, 1 (1965), 23–41.

Peter J Stuckey. 1991. Constructive negation for constraint logic programming. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 328–339.

Allen Van Gelder, Kenneth A Ross, and John S Schlipf. 1991. The well-founded semantics for general logic programs. *Journal of the ACM (JACM)* 38, 3 (1991), 619–649.

Certified Semantics for miniKanren*

DMITRY ROZPLOKHAS, Higher School of Economics and JetBrains Research, Russia
ANDREY VYATKIN, Saint Petersburg State University, Russia
DMITRY BOULYTCHEV, Saint Petersburg State University and JetBrains Research, Russia

We present two formal semantics for the core miniKanren. First, we give denotational variant which corresponds to the minimal Herbrand model for definite logic programs. Second, we present operational semantics which models interleaving, and prove its soundness and completeness w.r.t. denotational semantics. Our development is supported by formal Coq specification, thus making it certified.

CCS Concepts: • **Theory of computation → Constraint and logic programming**; **Denotational semantics**; **Operational semantics**;

Additional Key Words and Phrases: Relational programming, denotational semantics, operational semantics, certified programming

1 INTRODUCTION

The introductory book on miniKanren [Friedman et al. 2005] describes the language by means of an evolving set of examples. In the series of follow-up papers [Alvis et al. 2011; Byrd and Friedman 2007; Hemann and Friedman 2013, 2015; Hemann et al. 2016; Swords and Friedman 2013] various extensions of the language were presented with their semantics explained in terms of Scheme implementation. We argue that this style of semantic definition is fragile and not self-evident since it requires the knowledge of semantics of concrete implementation language. In addition the justification of important properties of relational programs (for example, refutational completeness [Byrd 2009]) becomes cumbersome. In the area of programming languages research a formal definition for the semantics of language of interest is a *de-facto* standard, and in our opinion in its current state miniKanren deviates from this standard.

There were some previous attempts to define a formal semantics for miniKanren. Lozov et al. [2017] present a variant of nondeterministic operational semantics, and Rozplokhas and Boulytchev [2018] use another variant of finite-set semantics. None of them was capable of reflecting the distinctive property of miniKanren search — *interleaving* [Kiselyov et al. 2005], thus deviating from the conventional understanding of the language.

In this paper we present a formal semantics for core miniKanren and prove some its basic properties. First, we define denotational semantics similar to the least Herbrand model for definite logic programs [Lloyd 1984]; then we describe operational semantics with interleaving in terms of labeled transition system. Finally, we prove the soundness and completeness of the operational semantics w.r.t the denotational one. We support our development

*This work was partially suppored by the grant 18-01-00380 from The Russian Foundation for Basic Research

Authors' addresses: Dmitry Rozplokhas, Higher School of Economics, JetBrains Research, Russia, darozplokhas@edu.hse.ru; Andrey Vyatkin, Saint Petersburg State University, Russia, dewshick@gmail.com; Dmitry Boulytchev, Saint Petersburg State University, JetBrains Research, Russia, dboulytchev@math.spbu.ru.

miniKanren.org/workshop/2019/8-ART5

$$
\begin{array}{lcll}
\mathcal{C} & = & \{C_i^{k_i}\} & \text{constructors with arities} \\
\mathcal{T}_X & = & X \cup \{C_i^{k_i}(t_1, \ldots, t_{k_i}) \mid t_j \in \mathcal{T}_X\} & \text{terms over the set of variables } X \\
\mathcal{D} & = & \mathcal{T}_\varnothing & \text{ground terms} \\
\mathcal{X} & = & \{x, y, z, \ldots\} & \text{syntactic variables} \\
\mathcal{A} & = & \{\alpha, \beta, \gamma, \ldots\} & \text{semantic variables} \\
\mathcal{R} & = & \{R_i^{k_i}\} & \text{relational symbols with arities} \\
\mathcal{G} & = & \mathcal{T}_\mathcal{X} \equiv \mathcal{T}_\mathcal{X} & \text{unification} \\
 & & \mathcal{G} \wedge \mathcal{G} & \text{conjunction} \\
 & & \mathcal{G} \vee \mathcal{G} & \text{disjunction} \\
 & & \textbf{fresh } \mathcal{X} \text{ . } \mathcal{G} & \text{fresh variable introduction} \\
 & & R_i^{k_i}(t_1, \ldots, t_{k_i}),\ t_j \in \mathcal{T}_\mathcal{X} & \text{relational symbol invocation} \\
\mathcal{S} & = & \{R_i^{k_i} = \lambda\, x_1^i \ldots x_{k_i}^i \,.\, g_i;\}\ g & \text{specification}
\end{array}
$$

Fig. 1. The syntax of the source language

$$
\begin{gathered}
\mathcal{FV}(x) = \{x\} \\
\mathcal{FV}(C_i^{k_i}(t_1, \ldots, t_k)) = \bigcup \mathcal{FV}(t_i) \\
\mathcal{FV}(t_1 \equiv t_2) = \mathcal{FV}(t_1) \cup \mathcal{FV}(t_2) \\
\mathcal{FV}(g_1 \wedge g_2) = \mathcal{FV}(g_1) \cup \mathcal{FV}(g_2) \\
\mathcal{FV}(g_1 \vee g_2) = \mathcal{FV}(g_1) \cup \mathcal{FV}(g_2) \\
\mathcal{FV}(\textbf{fresh } x \text{ . } g) = \mathcal{FV}(g) \setminus \{x\} \\
\mathcal{FV}(R_i^{k_i}(t_1, \ldots, t_k)) = \bigcup \mathcal{FV}(t_i)
\end{gathered}
$$

Fig. 2. Free variables in terms and goals

with a formal specification using Coq [Bertot and Castéran 2004] proof assistant[1], thus outsourcing the burden of proof checking to the automatic tool.

The paper organized as follows. In Section 2 we give the syntax of the language, describe its semantics informally and discuss some examples. Section 3 contains the description of denotational semantics for the language, and Section 4 — the operational semantics. In Section 5 we overview the certified proof for soundness and completeness of operational semantics. The final section concludes.

2 THE LANGUAGE

In this section we introduce the syntax of the language we use throughout the paper, describe the informal semantics and give some examples.

The syntax of the language is shown on Figure 1. First, we fix a set of constructors C with known arities and consider a set of terms $\mathcal{T}_X$ with constructors as functional symbols and variables from X. We parameterize this set with an alphabet of variables, since in the semantic description we will need *two* kinds of variables. The first kind, *syntactic* variables, is denoted by $\mathcal{X}$. We also consider an alphabet of *relational symbols* $\mathcal{R}$ which are used to name relational definitions. The central syntactic category in the language is a *goal*. In our case there are

[1] https://github.com/dboulytchev/miniKanren-coq

five types of goals: *unification* of terms, conjunction and disjunction of goals, fresh variable introduction and invocation of some relational definition. Thus, unification is used as a constraint, and multiple constraints can be combined using conjunction, disjunction and recursion. For the sake of brevity we abbreviate immediately nested "**fresh**" constructs into the one, writing "**fresh** $x\ y\ldots.\ g$" instead of "**fresh** x . **fresh** y . $\ldots g$". The final syntactic category is *specification* $\mathcal{S}$. It consists of a set of relational definitions and a top-level goal. A top-level goal represents a search procedure which returns a stream of substitutions for the free variables of the goal. The definition for set of free variables for both terms and goals is given on Figure 2; as "**fresh**" is the sole binding construct the definition is rather trivial. The language we defined is first-order, as goals can not be passed as parameters, returned or constructed at runtime.

We now informally describe how relational search works. As we said, a goal represents a search procedure. This procedure takes a *state* as input and returns a stream of states; a state (among other information) contains a substitution which maps semantic variables into terms over semantic variables. Then five types of scenarios are possible (dependent on the type of the goal):

- Unification "$t_1 \equiv t_2$" unifies terms t_1 and t_2 in the context of the substitution in the current state. If terms are unifiable, then their MGU is integrated into the substitution, and one-element stream is returned; otherwise the result is an empty stream.
- Conjunction "$g_1 \wedge g_2$" applies g_1 to the current state and then applies g_2 to the each element of the result, concatenating the streams.
- Disjunction "$g_1 \vee g_2$" applies both its goals to the current state independently and then concatenates the results.
- Fresh construct "**fresh** x . g" allocates a new semantic variable α, substitutes all free occurrences of x in g with α, and runs the goal.
- Invocation "$R_i^{k_i}\ (t_1,\ldots,t_{k_i})$" finds a definition for relational symbol $R_i^{k_i} = \lambda x_1 \ldots x_{k_i} . g_i$, substitutes all free occurrences of formal parameter x_j in g_i with term t_j (for all j) and runs the goal in the current state.

We stipulate, that the top-level goal is preceded by an implicit "**fresh**" construct, which binds all its free variables, and the final substitutions for these variables constitute the result of the goal evaluation.

Conjunction and disjunction form a monadic [Wadler 1995] interface with conjunction playing role of "bind" and disjunction — of "mplus". In this description we swept a lot of important details under the carpet — for example, in actual implementations the components of disjunction are not evaluated in isolation, but both disjuncts are being evaluated incrementally with the control passing from one disjunct to another (*interleaving*); instead streams the implementation can be based on "ferns" [Byrd et al. [n. d.]] to defer divergent computations, etc.

As an example consider the following specification:

```
append^o = λ x y xy .
  ((x ≡ Nil) ∧ (xy ≡ y)) ∨
  (fresh h t ty .
     (x  ≡ Cons (h, t))  ∧
     (xy ≡ Cons (h, ty)) ∧
     (append^o y t ty)
  );
revers^o = λ x y .
  ((x ≡ Nil) ∧ (y ≡ Nil)) ∨
  (fresh h t t' .
     (x ≡ Cons (h, t)) ∧
```

```
      (appendᵒ t' (Cons (h, Nil)) y) ∧
      (reversᵒ t t')
  );
reversᵒ x x
```

Here we defined two relational symbols — "appendᵒ" and "reversᵒ", — and specified a top-level goal "reversᵒ x x". The symbol "appendᵒ" defines a relational concatenation of lists — it takes three arguments and performs a case analysis on the first one. If the first one is an empty list ("Nil"), then the second and the third arguments are unified. Otherwise the first argument is deconstructed into a head "h" and a tail "t", and the tail is concatenated with the second argument using a recursive call to "appendᵒ" and additional variable "ty", which represents the concatenation of "t" and "y". Finally, we unify "Cons (h, ty)" with "xy" to form a final constraint. Similarly, "reversᵒ" defines relational list reversing. The top-level goal represents a search procedure for all lists "x", which are stable under reversing, i.e. represent palindromes. Running it results in an infinite stream of substitutions:

```
α ↦ Nil
α ↦ Cons (β₀, Nil)
α ↦ Cons (β₀, Cons (β₀, Nil))
α ↦ Cons (β₀, Cons (β₁, Cons (β₀, Nil)))
...
```

where "α" — a *semantic* variable, corresponding to "x", "β_i" — free semantics variables.

The notions above can be formalized in Coq in a natural way using inductive data types. We have made a few non-essential simplifications and modifications for the sake of convenience.

Specifically, we restrict the arities of constructors to be either zero or two:

```
Inductive term : Set :=
| Var : var → term
| Cst : con_name → term
| Con : con_name → term → term → term.
```

Here "var" and "con_name" — types representing variables and constructor names, whose definitions we omitted for the sake of brevity. Similarly, we restrict relations to always have exactly one argument:

```
Definition rel : Set := term → goal.
```

These restrictions do not make the language less expressive in any way since we can represent a sequence of terms as a list using constructors $\texttt{Nil}^0$ and $\texttt{Cons}^2$.

We also introduce one additional type of goals — *failure* — for deliberately unsuccessful computation (empty stream). As a result, the definition of goals looks as follows:

```
Inductive goal : Set :=
| Fail   : goal
| Unify  : term → term → goal
| Disj   : goal → goal → goal
| Conj   : goal → goal → goal
| Fresh  : (var → goal) → goal
| Invoke : rel_name → term → goal.
```

$$
\begin{array}{rcll}
x\,[t/x] & = & t & \\
y\,[t/x] & = & y & y \neq x \\
C_i^{k_i}\,(t_1, \ldots, t_{k_i})\,[t/x] & = & C_i^{k_i}\,(t_1\,[t/x], \ldots, t_{k_i}\,[t/x]) & \\
(t_1 \equiv t_2)\,[t/x] & = & t_1\,[t/x] \equiv t_2\,[t/x] & \\
(g_1 \wedge g_2)\,[t/x] & = & g_1\,[t/x] \wedge g_2\,[t/x] & \\
(g_1 \vee g_2)\,[t/x] & = & g_1\,[t/x] \vee g_2\,[t/x] & \\
(\textbf{fresh}\ x\,.\,g)\,[t/x] & = & \textbf{fresh}\ x\,.\,g & \\
(\textbf{fresh}\ y\,.\,g)\,[t/x] & = & \textbf{fresh}\ y\,.\,(g\,[t/x]) & y \neq x \\
(R_i^{k_i}\,(t_1, \ldots, t_{k_i})\,[t/x] & = & R_i^{k_i}\,(t_1\,[t/x], \ldots, t_{k_i}\,[t/x]) &
\end{array}
$$

Fig. 3. Substitutions for terms and goals

Note that in our formalization we use the higher-order abstract syntax for variable binding [Pfenning and Elliott 1988]. We preferred it to the first-order syntax because it gives us the ability to use substitution and inductive principle provided by Coq. However, we still need to carefully ensure some expected properties on the structure of syntax trees. For example, we should require that the definitions of relations do not contain unbound variables:

```
Definition closed_goal_in_context (c : list var) (g : goal) : Prop :=
  ∀ n, is_fv_of_goal n g → In n c.

Definition closed_rel (r : rel) : Prop :=
  ∀ (arg : term), closed_goal_in_context (fv_term arg) (r arg).

Definition def : Set := {r : rel | closed_rel r}.
```

In the snippet above "def" corresponds to a set of relational symbol definitions in a specification.

We set an arbitrary environment (a map from relational symbol to the definition of relation) to use further throughout the formalization. Failure goals allow us to define it as a total function:

```
Definition env : Set := rel_name → def.

Variable Prog : env.
```

3 DENOTATIONAL SEMANTICS

In this section we present a denotational semantics for the language we defined above. We use a simple set-theoretic approach which can be considered as an analogy to the least Herbrand model for definite logic programs [Lloyd 1984]. Strictly speaking, instead of developing it from scratch we could have just described the conversion of specifications into definite logic form and took their least Herbrand model. However, in that case we would still need to define the least Herbrand model semantics for definite logic programs in a certified way. In addition, while for this concrete language the conversion to definite logic form is trivial, it may become less trivial for its extensions (with, for examples, nominal constructs [Byrd and Friedman 2007]) which we plan to do in future.

To motivate further development, we first consider the following example. Let us have the following goal:

`x ≡ Cons (y, z)`

There are three free variables, and solving the goal delivers us the following single answer:

$\alpha \mapsto$ `Cons (`β`, `γ`)`

where semantic variables α, β and γ correspond to the syntactic ones "x", "y", "z". The goal does not put any constraints on "y" and "z", so there are no bindings for "β" and "γ" in the answer. This answer can be seen as the following ternary relation over the set of all ground terms:

$$\{(\texttt{Cons}\ (\beta,\gamma),\beta,\gamma) \mid \beta \in \mathcal{D},\ \gamma \in \mathcal{D}\} \subset \mathcal{D}^3$$

The order of "dimensions" is important, since each dimension corresponds to a certain free variable. Our main idea is to represent this relation as a set of total functions

$$\bar{\mathfrak{f}} : \mathcal{A} \mapsto \mathcal{D}$$

from semantic variables to ground terms. We call these functions *representing functions*. Thus, we may reformulate the same relation as

$$\{(\bar{\mathfrak{f}}(\alpha), \bar{\mathfrak{f}}(\beta), \bar{\mathfrak{f}}(\gamma)) \mid \bar{\mathfrak{f}} \in [\![\alpha \equiv \texttt{Cons}\ (\beta,\gamma)]\!]\}$$

where we use conventional semantic brackets "$[\![\bullet]\!]$" to denote the semantics. For the top-level goal we need to substitute its free syntactic variables with distinct semantic ones, calculate the semantics, and build the explicit representation for the relation as shown above. The relation, obviously, does not depend on concrete choice of semantic variables, but depends on the order in which the values of representing functions are tupled. This order can be conventionalized, which gives us a completely deterministic semantics.

Now we implement this idea. First, for a representing function

$$\bar{\mathfrak{f}} : \mathcal{A} \to \mathcal{D}$$

we introduce its homomorphic extension

$$\bar{\bar{\mathfrak{f}}} : \mathcal{T}_{\mathcal{A}} \to \mathcal{D}$$

which maps terms to terms:

$$\begin{array}{rcl} \bar{\bar{\mathfrak{f}}}(\alpha) & = & \bar{\mathfrak{f}}(\alpha) \\ \bar{\bar{\mathfrak{f}}}(C_i^{k_i}(t_1, \ldots . t_{k_i})) & = & C_i^{k_i}(\bar{\bar{\mathfrak{f}}}(t_1), \ldots \bar{\bar{\mathfrak{f}}}(t_{k_i})) \end{array}$$

Let us have two terms $t_1, t_2 \in \mathcal{T}_{\mathcal{A}}$. If there is a unifier for t_1 and t_2 then, clearly, there is a substitution θ which turns both t_1 and t_2 into the same *ground* term (we do not require θ to be the most general). Thus, θ maps (some) ground variables into ground terms, and its application to $t_{1(2)}$ is exactly $\bar{\bar{\theta}}(t_{1(2)})$. This reasoning can be performed in the opposite direction: a unification $t_1 \equiv t_2$ defines the set of all representing functions $\bar{\mathfrak{f}}$ for which $\bar{\bar{\mathfrak{f}}}(t_1) = \bar{\bar{\mathfrak{f}}}(t_2)$.

Then, the semantic function for goals is parameterized over environments which prescribe semantic functions to relational symbols:

$$\Gamma : \mathcal{R} \to (\mathcal{T}_{\mathcal{A}}{}^* \to 2^{\mathcal{A} \to \mathcal{D}})$$

An environment associates with relational symbol a function which takes a string of terms (the arguments of the relation) and returns a set of representing functions. The signature for semantic brackets for goals is as follows:

$$[\![\bullet]\!]_\Gamma : \mathcal{G} \to 2^{\mathcal{A} \to \mathcal{D}}$$

It maps a goal into the set of representing functions w.r.t. an environment Γ.

We formulate the following important *completeness condition* for the semantics of a goal g:

$$\forall \alpha \notin FV(g)\ \forall d \in \mathcal{D}\ \forall \mathfrak{f} \in [\![g]\!]\ \exists \mathfrak{f}' \in [\![g]\!]\ :\ \mathfrak{f}'(\alpha) = d \wedge \forall \beta \neq \alpha :\ \mathfrak{f}'(\beta) = \mathfrak{f}(\beta)$$

In other words, representing functions for a goal g restrict only the values of free variables of g and do not introduce any "hidden" correlations. This condition guarantees that our semantics is complete in the sense that it does not introduce artificial restrictions for the relation it defines. It can be proven that the semantics of goals always satisfy this condition.

We remind conventional notions of pointwise modification of a function

$$f[x \leftarrow v](z) = \begin{cases} f(z) & , \quad z \neq x \\ v & , \quad z = x \end{cases}$$

and substitution of a free variable with a term in terms and goals (see Figure 3).

For a representing function $\mathfrak{f} : \mathcal{A} \to \mathcal{D}$ and a semantic variable α we define the following *generalization* operation:

$$\mathfrak{f} \uparrow \alpha = \{\mathfrak{f}[\alpha \leftarrow d] \mid d \in \mathcal{D}\}$$

Informally, this operation generalizes a representing function into a set of representing functions in such a way that the values of these functions for a given variable cover the whole $\mathcal{D}$. We extend the generalization operation for sets of representing functions $\mathfrak{F} \subseteq \mathcal{A} \to \mathcal{D}$:

$$\mathfrak{F} \uparrow \alpha = \bigcup_{\mathfrak{f} \in \mathfrak{F}} (\mathfrak{f} \uparrow \alpha)$$

Now we are ready to specify the semantics for goals (see Figure 4). We've already given the motivation for the semantics of unification: the condition $\bar{\mathfrak{f}}(t_1) = \bar{\mathfrak{f}}(t_2)$ gives us the set of all (otherwise unrestricted) representing functions which "equate" terms t_1 and t_2. Set union and intersection provide a conventional interpretation for disjunction and conjunction of goals, and the semantics of relational invocation reduces to the application of corresponding function from the environment. The only interesting case is "**fresh** x . g". First, we take an arbitrary semantic variable α, not free in g, and substitute x with α. Then we calculate the semantics of $g[\alpha/x]$. The interesting part is the next step: as x can not be free in "**fresh** x . g", we need to generalize the result over α since in our model the semantics of a goal specifies a relation over its free variables. We introduce some nondeterminism, by choosing arbitrary α, but it can be proven by structural induction, that with different choices of free variable, semantics of a goal won't change. Consider the following example:

fresh `y` . $(\alpha \equiv$ `y`$)\ \wedge\ ($`y` $\equiv$ `Zero`$)$

As there is no invocations involved, we can safely omit the environment. Then:

$$
\begin{array}{rcll}
[\![t_1 \equiv t_2]\!]_\Gamma & = & \{\bar{f} : \mathcal{A} \to \mathcal{D} \mid \bar{f}(t_1) = \bar{f}(t_2)\} & [\textsc{Unify}_D] \\
[\![g_1 \wedge g_2]\!]_\Gamma & = & [\![g_1]\!]_\Gamma \cap [\![g_1]\!]_\Gamma & [\textsc{Conj}_D] \\
[\![g_1 \vee g_2]\!]_\Gamma & = & [\![g_1]\!]_\Gamma \cup [\![g_1]\!]_\Gamma & [\textsc{Dis}_D] \\
[\![\textbf{fresh}\ x\ .\ g]\!]_\Gamma & = & ([\![g\,[\alpha/x]]\!]_\Gamma) \uparrow \alpha,\ \alpha \notin FV(g) & [\textsc{Fresh}_D] \\
[\![R(t_1, \ldots, t_k)]\!]_\Gamma & = & (\Gamma\, R)\, t_1 \ldots t_k & [\textsc{Invoke}_D]
\end{array}
$$

Fig. 4. Denotational semantics of goals

$$
\begin{array}{lcl}
[\![\textbf{fresh}\ \mathtt{y}\ .\ (\alpha \equiv \mathtt{y}) \wedge (\mathtt{y} \equiv \mathtt{Zero})]\!] & = & \text{(by } \textsc{Fresh}_D\text{)} \\
([\![(\alpha \equiv \beta) \wedge (\beta \equiv \mathtt{Zero})]\!]) \uparrow \beta & = & \text{(by } \textsc{Conj}_D\text{)} \\
([\![\alpha \equiv \beta]\!] \cap [\![\beta \equiv \mathtt{Zero})]\!]) \uparrow \beta & = & \text{(by } \textsc{Unify}_D\text{)} \\
(\{\bar{f} \mid \bar{f}(\alpha) = \bar{f}(\beta)\} \cap \{\bar{f} \mid \bar{f}(\beta) = \bar{f}(\mathtt{Zero})\}) \uparrow \beta & = & \text{(by the definition of “}\bar{f}\text{”)} \\
(\{\bar{f} \mid \bar{f}(\alpha) = \bar{f}(\beta)\} \cap \{\bar{f} \mid \bar{f}(\beta) = \mathtt{Zero}\}) \uparrow \beta & = & \text{(by the definition of “}\cap\text{”)} \\
(\{\bar{f} \mid \bar{f}(\alpha) = \bar{f}(\beta) = \mathtt{Zero}\}) \uparrow \beta & = & \text{(by the definition of “}\uparrow\text{”)} \\
\{\bar{f} \mid \bar{f}(\alpha) = \mathtt{Zero}, \bar{f}(\beta) = d, d \in \mathcal{D}\} & = & \text{(by the totality of representing functions)} \\
\{\bar{f} \mid \bar{f}(\alpha) = \mathtt{Zero}\} &&
\end{array}
$$

In the end we've got the set of representing functions, each of which restricts only the value of free variable α.

The final component is the semantics of specifications. Given a specification

$$\{R_i = \lambda\, x_1^i \ldots x_{k_i}^i\ .\ g_i;\}_{i=1}^n\ g$$

we have to construct a correct environment Γ_0 and then take the semantics of the top-level goal:

$$[\![\{R_i = \lambda\, x_1^i \ldots x_{k_i}^i\ .\ g_i;\}_{i=1}^n\ g]\!] = [\![g]\!]_{\Gamma_0}$$

As the set of definitions can be mutually recursive we apply the fixed point approach. We consider the following function

$$\mathcal{F} : (\mathcal{T}_\mathcal{A}^* \to 2^{\mathcal{A} \to \mathcal{D}})^n \to (\mathcal{T}_\mathcal{A}^* \to 2^{\mathcal{A} \to \mathcal{D}})^n$$

which represents a semantic for the set of definitions abstracted over themselves. The definition of this function is rather standard:

$$
\begin{array}{rcl}
\mathcal{F}(p_1, \ldots, p_n) & = & (t_1^1 \ldots t_{k_1}^1 \mapsto [\![g^1\,[t_1^1/x_1^1, \ldots, t_{k_1}^1/x_{k_1}^1]]\!]_\Gamma, \\
& & \ldots \\
& & t_1^n \ldots t_{k_n}^n \mapsto [\![g^n\,[t_1^n/x_1^n, \ldots, t_{k_n}^n/x_{k_n}^n]]\!]_\Gamma) \\
& & \text{where } \Gamma\, R_i = p_i
\end{array}
$$

Here p_i is a semantic function for i-th definition; we build an environment Γ which associates each relational symbol R_i with p_i and construct a n-dimensional vector-function, where i-th component corresponds to a function which calculates the semantics of i-th relational definition application to terms w.r.t. the environment Γ. Finally, we take the least fixed point of $\mathcal{F}$ and define the top-level environment as follows:

$$\Gamma_0\, R_i = (fix\ \mathcal{F})\,[i]$$

where "[i]" denotes the i-th component of a vector-function.

The least fixed point exists by Knaster-Tarski [Tarski 1955] theorem — the set $(\mathcal{T}_{\mathcal{A}}{}^{*} \to 2^{\mathcal{A}\to\mathcal{D}})^{n}$ forms a complete lattice, and $\mathcal{F}$ is monotonic.

To formalize denotational semantics in Coq we can define representing functions simply as Coq functions:

```
Definition repr_fun : Set := var → ground_term.
```

We define the semantics via inductive proposition "in_denotational_sem_goal" such that

$$\forall g, \mathfrak{f} \ : \ \texttt{in\_denotational\_sem\_goal}\ g\ \mathfrak{f} \Longleftrightarrow \mathfrak{f} \in [\![g]\!]_{\Gamma}$$

The definition is as follows:

```
Inductive in_denotational_sem_goal : goal → repr_fun → Prop :=
| dsgUnify  : ∀ f t1 t2,  apply_repr_fun f t1 = apply_repr_fun f t2 →
                          in_denotational_sem_goal (Unify t1 t2) f

| dsgDisjL  : ∀ f g1 g2,  in_denotational_sem_goal g1 f →
                          in_denotational_sem_goal (Disj g1 g2) f

| dsgDisjR  : ∀ f g1 g2,  in_denotational_sem_goal g2 f →
                          in_denotational_sem_goal (Disj g1 g2) f

| dsgConj   : ∀ f g1 g2,  in_denotational_sem_goal g1 f →
                          in_denotational_sem_goal g2 f →
                          in_denotational_sem_goal (Conj g1 g2) f

| dsgFresh  : ∀ f fn a fg,  (~ is_fv_of_goal a (Fresh fg)) →
                            in_denotational_sem_goal (fg a) fn →
                            (∀ x, x <> a → fn x = f x) →
                            in_denotational_sem_goal (Fresh fg) f

| dsgInvoke : ∀ r t f,  in_denotational_sem_goal (proj1_sig (Prog r) t) f →
                        in_denotational_sem_goal (Invoke r t) f.
```

Here we refer to a fixpoint "apply_repr_fun" which calculates the extension "$\overline{\bullet}$" for a representing function, and inductive proposition "is_fv_of_goal" which encodes the set of free variables for a goal.

Recall that the environment "Prog" maps every relational symbol to the definition of relation, which is a pair of a function from terms to goals and a proof that it has no unbound variables. So in the last case "(proj1_sig (Prog r) t) " simply takes the body of the corresponding relation; thus "Prog" in Coq specification plays role of a global environment Γ.

It is interesting that in Coq implementation we do not need to refer to Tarski-Knaster theorem explicitly since the least fixpoint semantic is implicitly provided by inductive definitions.

4 OPERATIONAL SEMANTICS

In this section we describe operational semantics of miniKanren, which corresponds to the known implementations with interleaving search. The semantics will be given in the form of labeled transition system (LTS). From now on we assume the set of semantic variables to be linearly ordered ($\mathcal{A} = \{\alpha_1, \alpha_2, \dots\}$).

We introduce the notion of substitution

$$\sigma : \mathcal{A} \to \mathcal{T}_{\mathcal{A}}$$

as a (partial) mapping from semantic variables to terms over the set of semantic variables. We denote Σ the set of all substitutions, $\mathcal{D}om(\sigma)$ — the domain for a substitution σ, $\mathcal{VR}an(\sigma) = \bigcup_{\alpha \in \mathcal{D}om(\sigma)} \mathcal{FV}(\sigma(\alpha))$ — its range (the set of all free variables in the image).

The states in the transition system have the following shape

$$S = \mathcal{G} \times \Sigma \times \mathbb{N} \mid S \oplus S \mid S \otimes \mathcal{G}$$

As we will see later, an evaluation of a goal is separated into elementary steps, and these steps are performed interchangeably for different subgoals. Thus, a state has a tree-like structure with intermediate nodes corresponding to partially-evaluated conjunctions ("$\otimes$") or disjunctions ("$\oplus$"). A leaf in the form $\langle g, \sigma, n\rangle$ determines a goal in a context, where g — a goal, σ — a substitution accumulated so far, and n — a natural number, which corresponds to a number of semantic variables used to this point. For a conjunction node its right child is always a goal since it cannot be evaluated unless some result is provided by the left conjunct.

We also need extended states

$$\overline{S} = \diamond \mid S$$

where $\diamond$ symbolizes the end of evaluation, and the following well-formedness condition:

Definition 1. *Well-formedness condition for extended states:*

- $\diamond$ *is well-formed;*
- $\langle g, \sigma, n\rangle$ *is well-formed iff* $\mathcal{FV}(g) \cup \mathcal{D}om(\sigma) \cup \mathcal{VR}an(\sigma) \subset \{\alpha_1, \dots, \alpha_n\}$*;*
- $s_1 \oplus s_2$ *is well-formed iff* s_1 *and* s_2 *well-formed;*
- $s \otimes g$ *is well-formed iff* s *is well-formed and for all leaf triplets* $\langle \_, \_, n\rangle$ *in* s $\mathcal{FV}(g) \subseteq \{\alpha_1, \dots, \alpha_n\}$.

Informally the well-formedness restricts the set of states to those in which all goals use only allocated variables.

Finally, we define the set of labels:

$$L = \circ \mid \Sigma \times \mathbb{N}$$

The label "$\circ$" is used to mark those steps which do not provide an answer; otherwise a transition is labeled by a pair of a substitution and a number of allocated variables. The substitution is one of the answers, and the number is threaded through the derivation to keep track of allocated variables; we ignore it in further explanations.

The transition rules are shown on Figure 5. The first two rules specify the semantics of unification. If two terms are not unifiable under the current substitution σ then the evaluation stops with no answer; otherwise it stops with the answer equal to the most general unifier.

The next two rules describe the steps performed when disjunction (conjunction) is encountered on the top level of the current goal. For disjunction it schedules both goals (using "$\oplus$") for evaluating in the same context as the parent state, for conjunction — schedules the left goal and postpones the right one (using "$\otimes$").

The rule for "**fresh**" substitutes bound syntactic variable with a newly allocated semantic one and proceeds with the goal; no answer provided at this step.

$$\langle t_1 \equiv t_2, \sigma, n\rangle \xrightarrow{\circ} \diamond,\ \nexists\, mgu\,(t_1, t_2, \sigma) \qquad \text{[UnifyFail]}$$

$$\langle t_1 \equiv t_2, \sigma, n\rangle \xrightarrow{(mgu\,(t_1, t_2, \sigma),\, n)} \diamond \qquad \text{[UnifySuccess]}$$

$$\langle g_1 \vee g_2, \sigma, n\rangle \xrightarrow{\circ} \langle g_1, \sigma, n\rangle \oplus \langle g_2, \sigma, n\rangle \qquad \text{[Disj]}$$

$$\langle g_1 \wedge g_2, \sigma, n\rangle \xrightarrow{\circ} \langle g_1, \sigma, n\rangle \otimes g_2 \qquad \text{[Conj]}$$

$$\langle \textbf{fresh}\ x\ .\ g, \sigma, n\rangle \xrightarrow{\circ} \langle g\,[\alpha_{n+1}\,/_x\,], \sigma, n+1\rangle \qquad \text{[Fresh]}$$

$$\frac{R_i^{k_i} = \lambda\, x_1 \ldots x_{k_i}\ .\ g}{\left\langle R_i^{k_i}\,(t_1, \ldots, t_{k_i}), \sigma, n\right\rangle \xrightarrow{\circ} \left\langle g\,[t_1\,/_{x_1}\,\ldots\,t_{k_i}\,/_{x_{k_i}}\,], \sigma, n\right\rangle} \qquad \text{[Invoke]}$$

$$\frac{s_1 \xrightarrow{\circ} \diamond}{(s_1 \oplus s_2) \xrightarrow{\circ} s_2} \qquad \text{[DisjStop]}$$

$$\frac{s_1 \xrightarrow{r} \diamond}{(s_1 \oplus s_2) \xrightarrow{r} s_2} \qquad \text{[DisjStopAns]}$$

$$\frac{s \xrightarrow{\circ} \diamond}{(s \otimes g) \xrightarrow{\circ} \diamond} \qquad \text{[ConjStop]}$$

$$\frac{s \xrightarrow{(\sigma, n)} \diamond}{(s \otimes g) \xrightarrow{\circ} \langle g, \sigma, n\rangle} \qquad \text{[ConjStopAns]}$$

$$\frac{s_1 \xrightarrow{\circ} s_1'}{(s_1 \oplus s_2) \xrightarrow{\circ} (s_2 \oplus s_1')} \qquad \text{[DisjStep]}$$

$$\frac{s_1 \xrightarrow{r} s_1'}{(s_1 \oplus s_2) \xrightarrow{r} (s_2 \oplus s_1')} \qquad \text{[DisjStepAns]}$$

$$\frac{s \xrightarrow{\circ} s'}{(s \otimes g) \xrightarrow{\circ} (s' \otimes g)} \qquad \text{[ConjStep]}$$

$$\frac{s \xrightarrow{(\sigma, n)} s'}{(s \otimes g) \xrightarrow{\circ} (\langle g, \sigma, n\rangle \oplus (s' \otimes g))} \qquad \text{[ConjStepAns]}$$

Fig. 5. Operational semantics of interleaving search

The rule for relation invocation finds a corresponding definition, substitutes its formal parameters with the actual ones, and proceeds with the body.

The rest of the rules specify the steps performed during the evaluation of two remaining types of the states — conjunction and disjunction. In all cases the left state is evaluated first. If its evaluation stops with a result then the right state (or goal) is scheduled for evaluation, and the label is propagated. If there is no result then the

conjunction evaluation stops with no result (ConjStop) as well while the disjunction evaluation proceeds with the right state (DisjStop).

The last four rules describe *interleaving*, which occurs when the evaluation of the left state suspends with some residual state (with or without an answer). In the case of disjunction the answer (if any) is propagated, and the constituents of the disjunction are swapped (DisjStep, DisjStepAns). In case of conjunction, if the evaluation step in the left conjunct did not provide any answer, the evaluation is continued in the same order since there is still no information to proceed with the evaluation of the right conjunct (ConjStep); if there is some answer, then the disjunction of the right conjunct in the context of the answer and the remaining conjunction is scheduled for evaluation (ConjStepAns).

The introduced transition system is completely deterministic. There was, however, some freedom in choosing the order of evaluation for conjunction and disjunction states. For example, instead of evaluating the left substate first we could choose to evaluate the right one, etc. In each concrete case we would end up with a different (but still deterministic) system which would prescribe different semantics to a concrete goal. This choice reflects the inherent non-deterministic nature of search in relational (and, more generally, logical) programming. However, as long as deterministic search procedures are sound and complete, we can consider them "equivalent"[2].

A derivation sequence for a certain state determines a *trace* — a finite or infinite sequence of answers. We may define a set of finite or infinite sequences X^ω over an alphabet X as a set of functions from natural numbers into a lifted set $X_\perp = X \cup \{\perp\}$:

$$X^\omega = \{\omega : \mathbb{N} \to X_\perp \mid \forall n \in \mathbb{N},\, \omega(n) = \perp \Rightarrow \omega(n+1) = \perp\}$$

Informally speaking, we represent a sequence as a function which maps positions (treated as natural numbers) into the elements of the sequence. We use "$\perp$" to specify that there is no element at given position, and we stipulate, that there are no "holes" in this representation: if there is no element at given position then there are no elements at greater positions as well.

For this representation we may define the empty sequence ϵ and operations of prepending a sequence ω with an element a and taking a suffix of a sequence ω from a position n as follows:

$$\epsilon = i \mapsto \perp$$

$$a\omega = i \mapsto \begin{cases} a & , \quad i = 0 \\ \omega(i-1) & , \quad \text{otherwise} \end{cases}$$

$$\omega[n:] = i \mapsto \omega(n+i)$$

For a given state s a trace $\mathcal{T}r_s \in L^\omega$ is a sequence of labels, defined as follows simultaneously with the sequence of states $\{s_i\}$:

$$s_o = s$$

$$\begin{array}{llll} \mathcal{T}r_s(n) = a & , \quad s_{n+1} = s' & \text{if} & s_n \neq \diamond,\, s_n \xrightarrow{a} x' \\ \mathcal{T}r_s(n) = \perp & , \quad s_{n+1} = \diamond & \text{if} & s_n = \diamond \end{array}$$

The trace corresponds to the stream of answers in the reference miniKanren implementations.

To formalize the operational part in Coq we first need to define all preliminary notions from unification theory [Baader and Snyder 2001] which our semantics uses.

[2]There still can be differences in observable behavior of concrete goals under different sound and complete search strategies: a goal can be refutationally complete [Byrd 2009] under one strategy and non-complete under another.

In particular, we need to implement the notion of the most general unifier (MGU). As is it well-known [McBride 2003] all standard recursive algorithms for calculating MGU are not decreasing on argument terms, so we can't define it as a simple recursive function in Coq due to the termination check. There is no such obstacle when we define MGU as a proposition:

```
Inductive MGU : term → term → option subst → Set := ...
```

However, we still need to use a well-founded induction to prove the existence of the most general unifier and its defining properties:

```
Lemma MGU_ex : ∀ t1 t2, {r & MGU t1 t2 r}.

Definition unifier (s : subst) (t1 t2 : term) : Prop := apply_subst s t1 = apply_subst s t2.

Lemma MGU_unifies:
  ∀ t1 t2 s, MGU t1 t2 (Some s) → unifier s t1 t2.

Definition more_general (m s : subst) : Prop :=
  ∃ (s' : subst), ∀ (t : term), apply_subst s t = apply_subst s' (apply_subst m t).

Lemma MGU_most_general :
  ∀ (t1 t2 : term) (m : subst),
    MGU t1 t2 (Some m) →
    ∀ (s : subst), unifier s t1 t2 → more_general m s.

Lemma MGU_non_unifiable :
  ∀ (t1 t2 : term),
    MGU t1 t2 None → ∀ s, ~ (unifier s t1 t2).
```

For this well-founded induction we use the number of free variables in argument terms as a well-founded order on pairs of terms:

```
Definition terms := term * term.

Definition fvOrder (t : terms) := length (union (fv_term (fst t)) (fv_term (snd t))).

Definition fvOrderRel (t p : terms) := fvOrder t < fvOrder p.

Lemma fvOrder_wf : well_founded fvOrderRel.
```

After this preliminary work, the described transition relation can be encoded naturally as an inductively defined proposition (here "`state'`" stands for an extended state):

```
Inductive eval_step : state → label → state' → Set := ...
```

We state the fact that our system is deterministic through existence and uniqueness of a transition for every state:

```
Lemma eval_step_ex : ∀ (st : state), {l : label & {st' : state' & eval_step st l st'}}.
```

```
Lemma eval_step_unique :
  ∀ (st : state) (l1 l2 : label) (st'1 st'2 : state'),
    eval_step st l1 st'1 → eval_step st l2 st'2 → l1 = l2 ∧ st'1 = st'2.
```

To work with (possibly) infinite sequences we use the standard approach in Coq — coinductively defined streams:

```
Context {A : Set}.

CoInductive stream : Set :=
| Nil : stream
| Cons : A → stream → stream.
```

Although the definition of the datatype is coinductive some of its properties we are working with make sense only when defined inductively:

```
Inductive in_stream : A → stream → Prop :=
| inHead : ∀ x t, in_stream x (Cons x t)
| inTail : ∀ x h t, in_stream x t → in_stream x (Cons h t).

Inductive finite : stream → Prop :=
| fNil : finite Nil
| fCons : ∀ h t, finite t → finite (Cons h t).
```

Then we define a trace coinductively as a stream of labels in transition steps and prove that there exists a unique trace from any extended state:

```
Definition trace : Set := @stream label.

CoInductive op_sem : state' → trace → Set :=
| osStop : op_sem Stop Nil
| osState : ∀ st l st' t, eval_step st l st' →
                          op_sem st' t →
                          op_sem (State st) (Cons l t).

Lemma op_sem_ex (st' : state') : {t : trace & op_sem st' t}.

Lemma op_sem_unique :
  ∀ st' t1 t2, op_sem st' t1 → op_sem st' t2 → equal_streams t1 t2.
```

Note, for the equality of streams we need to define a new coinductive proposition instead of using the standard syntactic equality in order for coinductive proofs to work [Chlipala 2013].

One thing we can prove using operational semantics is the *interleaving* properties of disjunction. Specifically, we can prove that a trace for a disjunction is a one-by-one interleaving of streams for its disjuncts:

```
CoInductive interleave : stream → stream → stream → Prop :=
| interNil : ∀ s s', equal_streams s s' → interleave Nil s s'
| interCons : ∀ h t s rs, interleave s t rs → interleave (Cons h t) s (Cons h rs).
```

```
Lemma sum_op_sem : ∀ st1 st2 t1 t2 t, op_sem (State st1) t1 →
                                      op_sem (State st2) t2 →
                                      op_sem (State (Sum st1 st2)) t →
                                      interleave t1 t2 t.
```

This allows us to prove the expected properties of interleaving in a more general setting of arbitrary streams:

- the elements of the interleaved stream are exactly those of two interleaved streams;
- the interleaved stream is finite iff both interleaving streams are finite.

The corresponding Coq lemmas are as follows:

```
Lemma interleave_in : ∀ s1 s2 s, interleave s1 s2 s →
                ∀ x, in_stream x s ↔ in_stream x s1 ∨ in_stream x s2.

Lemma interleave_finite : ∀ s1 s2 s, interleave s1 s2 s →
                (finite s ↔ finite s1 ∧ finite s2).
```

5 SEMANTICS EQUIVALENCE

Now when we defined two different kinds of semantics for miniKanren we can relate them and show that the results given by these two semantics are the same for any specification. This will actually say something important about the search in the language: since operational semantics describes precisely the behavior of the search and denotational semantics ignores the search and describes what we *should* get from mathematical point of view, by proving their equivalence we establish *completeness* of the search which means that the search will get all answers satisfying the described specification and only those.

But first, we need to relate the answers produced by these two semantics as they have different forms: a trace of substitutions (along with numbers of allocated variables) for operational and a set of representing functions for denotational. We can notice that the notion of representing function is close to substitution, with only two differences:

- representing function is total;
- terms in the domain of representing function are ground.

Therefore we can easily extend (perhaps ambiguously) any substitution to a representing function by composing it with an arbitrary representing function and that will preserve all variable dependencies in the substitution. So we can define a set of representing functions corresponding to substitution as follows:

$$[\sigma] = \{\bar{\mathfrak{f}} \circ \sigma \mid \mathfrak{f} : \mathcal{A} \mapsto \mathcal{D}\}$$

And *denotational analog* of an operational semantics (a set of representing functions corresponding to answers in the trace) for given extended state s is then defined as a union of sets for all substitution in the trace:

$$[\![s]\!]_{op} = \cup_{(\sigma,n)\in\mathcal{T}r_s}[\sigma]$$

This allows us to state theorems relating two semantics.

Theorem 1 (Operational semantics soundness). *For any specification* $\{\dots\}\ g$*, for which the indices of all free variables in* g *are limited by some number* n

$$[\![\langle g, \epsilon, n\rangle]\!]_{op} \subset [\![\{\dots\}\ g]\!].$$

$$\begin{array}{rcl} [\![\diamond]\!]_\Gamma & = & \varnothing \\ [\![\langle g, \sigma, n\rangle]\!]_\Gamma & = & [\![g]\!]_\Gamma \cap [\sigma] \\ [\![s_1 \oplus s_2]\!]_\Gamma & = & [\![s_1]\!]_\Gamma \cup [\![s_2]\!]_\Gamma \\ [\![s \otimes g]\!]_\Gamma & = & [\![s]\!]_\Gamma \cap [\![g]\!]_\Gamma \end{array}$$

Fig. 6. Denotational semantics of states

It can be proven by nested induction, but first, we need to generalize the statement so that the inductive hypothesis would be strong enough for the inductive step. To do so, we define denotational semantics not only for goals but for arbitrarily extended states. Note that this definition does not need to have any intuitive interpretation, it is introduced only for proof to go smoothly. The definition of the denotational semantics for extended states is on Figure 6. The generalized version of the theorem uses it:

Lemma 1 (Generalized soundness). *For any top-level environment Γ_0 acquired from some specification, for any well-formed (w.r.t. that specification) extended state s*

$$[\![s]\!]_{op} \subset [\![s]\!]_{\Gamma_0}.$$

It can be proven by induction on the number of steps in which a given answer (more accurately, the substitution that contains it) occurs in the trace. The induction step is proven by structural induction on the extended state s.

It would be tempting to formulate the completeness of operational semantics as the inverse inclusion, but it does not hold in such generality. The reason for this is that denotational semantics encodes only dependencies between the free variables of a goal, which is reflected by the completeness condition, while operational semantics may also contain dependencies between semantic variables allocated in "**fresh**". Therefore we formulate the completeness with representing functions restricted on the semantic variables allocated in the beginning (which includes all free variables of a goal). This does not compromise our promise to prove the completeness of the search as miniKanren provides the result as substitutions only for queried variables, which are allocated in the beginning.

Theorem 2 (Operational semantics completeness). *For any specification $\{\dots\}$ g, for which the indices of all free variables in g are limited by some number n*

$$\{\hat{\sigma}|_{\{\alpha_1,\dots,\alpha_n\}} \mid \hat{\sigma} \in [\![\{\dots\}\ g]\!]\} \subset \{\hat{\sigma}|_{\{\alpha_1,\dots,\alpha_n\}} \mid \hat{\sigma} \in [\![\langle g, \epsilon, n\rangle]\!]_{op}\}.$$

Similarly to the soundness, this can be proven by nested induction, but the generalization is required. This time it is enough to generalize it from goals to states of the shape $\langle g, \sigma, n\rangle$. We also need to introduce one more auxiliary semantics — bounded denotational semantics:

$$[\![\bullet]\!]^l : \mathcal{G} \to 2^{\mathcal{A} \to \mathcal{D}}$$

Instead of always unfolding the definition of a relation for invocation goal, it does so only given number of times. So for a given set of relational definitions $\{R_i^{k_i} = \lambda\, x_1^i \dots x_{k_i}^i\, .\, g_i;\}$ the definition of bounded denotational semantics is exactly the same as in usual denotational semantics, except that for the invocation case:

$$[\![R_i^{k_i}\,(t_1, \dots, t_{k_i})]\!]^{l+1} = [\![g_i[t_1/x_1^i, \dots, t_{k_i}/x_{k_i}^i]]\!]^l$$

It is convenient to define bounded semantics for level zero as an empty set:

$$[\![g]\!]^0 = \varnothing$$

Bounded denotational semantics is an approximation of a usual denotational semantics and it is clear that any answer in usual denotational semantics will also be in bounded denotational semantics for some level:

Lemma 2. $[\![g]\!]_{\Gamma_0} \subset \cup_l [\![g]\!]^l$

Formally it can be proven using the definition of the least fixed point from Tarski-Knaster theorem: the set on the right-hand side is a closed set.

Now the generalized version of the completeness theorem is as follows:

Lemma 3 (Generalized completeness). *For any set of relational definitions, for any level l, for any well-formed (w.r.t. that set of definitions) state $\langle g, \sigma, n\rangle$,*

$$\{\mathfrak{f}|_{\{\alpha_1,\ldots,\alpha_n\}} \mid \mathfrak{f} \in [\![g]\!]^l \cap [\sigma]\} \subset \{\mathfrak{f}|_{\{\alpha_1,\ldots,\alpha_n\}} \mid \mathfrak{f} \in [\![\langle g, \sigma, n\rangle]\!]_{op}\}.$$

It is proven by induction on the level l. The induction step is proven by structural induction on the goal g.

The proofs of both theorems are certified in Coq, although the proofs for a number of (obvious) technical facts about representing functions and computation of the most general unifier as well as some properties of denotational semantics, proven informally in Section 3, are admitted for now. For completeness we can not just use the induction on proposition `in_denotational_sem_goal`, as it would be natural to expect, because the inductive principle it provides is not flexible enough. So we need to define bounded denotational semantics in our formalization too and perform induction on the level explicitly:

```
Inductive in_denotational_sem_lev_goal : nat → goal → repr_fun → Prop :=
...
| dslgInvoke : ∀ l r t f,
    in_denotational_sem_lev_goal l (proj1_sig (Prog r) t) f →
    in_denotational_sem_lev_goal (S l) (Invoke r t) f.
```

The lemma relating bounded and unbounded denotational semantics is translated into Coq:

```
Lemma in_denotational_sem_some_lev: ∀ (g : goal) (f : repr_fun),
    in_denotational_sem_goal g f →
    ∃ l, in_denotational_sem_lev_goal l g f.
```

The statements of the theorems are as follows:

```
Theorem search_correctness: ∀ (g : goal) (k : nat) (f : repr_fun) (t : trace),
  closed_goal_in_context (first_nats k) g) →
  op_sem (State (Leaf g empty_subst k)) t) →
  in_denotational_analog t f →
  in_denotational_sem_goal g f.

Theorem search_completeness: ∀ (g : goal) (k : nat) (f : repr_fun) (t : trace),
  closed_goal_in_context (first_nats k) g) →
  op_sem (State (Leaf g empty_subst k)) t) →
  in_denotational_sem_goal g f →
  ∃ (f' : repr_fun), (in_denotational_analog t f') ∧
                     ∀ (x : var), In x (first_nats k) → f x = f' x.
```

One important immediate corollary of these theorems is the correctness of certain program transformations. Since the results obtained by the search on a specification are exactly the results from the mathematical model of this specification, after the transformations of relations that do not change their mathematical meaning the search will obtain the same results. Note that this way we guarantee only the stability of results as the set of ground terms, the other aspects of program behavior, such as termination, may be affected. This allows us to safely (to a certain extent) apply such natural transformations as:

- changing the order of constituents in conjunction or disjunction;
- swapping conjunction and disjunction using distributivity;
- moving fresh variable introduction.

and even transform relational definitions to some kinds of normal form (like all fresh variables introduction on the top level with the conjunctive normal form inside), which may be convenient, for example, for metacomputation.

6 CONCLUSION AND FUTURE WORK

In this paper we presented a formal semantics for core miniKanren and proved some its basic properties, which are believed to hold in existing implementations. We consider our work as an initial setup for a future development of miniKanren semantics. The language we considered here lacks many important features, which are already introduced and employed in many implementations. Integrating these extensions — in the first hand, disequality constraints, — into the semantics looks a natural direction for future work. We also are going to address the problems of proving some properties of relational programs (equivalence, refutational completeness, etc.).

REFERENCES

Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. 2011. cKanren: miniKanren with Constraints. In *Proceedings of the 2011 Annual Workshop on Scheme and Functional Programming*.

Franz Baader and Wayne Snyder. 2001. Handbook of Automated Reasoning. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, Chapter Unification Theory.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. Ph.D. Dissertation. Indiana University.

William E. Byrd and Daniel P. Friedman. 2007. αkanren: A Fresh Name in Nominal Logic Programming. In *Proceedings of the 2007 Annual Workshop on Scheme and Functional Programming*. 79–90.

William E. Byrd, Daniel P. Friedman, Ramana Kumar, and Joseph P. Near. [n. d.]. A Shallow Scheme Embedding of Bottom-Avoiding Streams. ([n. d.]).

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer*. The MIT Press.

Jason Hemann and Daniel P. Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming*.

Jason Hemann and Daniel P. Friedman. 2015. A Framework for Extending microKanren with Constraints. In *Proceedings of the 2015 Annual Workshop on Scheme and Functional Programming*.

Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2016. A Small Embedding of Logic Programming with a Simple Complete Search. *SIGPLAN Not.* 52, 2 (Nov. 2016), 96–107. https://doi.org/10.1145/3093334.2989230

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). *SIGPLAN Not.* 40, 9 (Sept. 2005), 192–203. https://doi.org/10.1145/1090189.1086390

J. W. Lloyd. 1984. *Foundations of Logic Programming*. Springer-Verlag, Berlin, Heidelberg.

Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2017. Typed Relational Conversion. In *Proceedings of the International Symposium on Trends in Functional Programming*.

Conor McBride. 2003. First-order Unification by Structural Recursion. *J. Funct. Program.* 13, 6 (Nov. 2003), 1061–1075. https://doi.org/10.1017/S0956796803004957

F. Pfenning and C. Elliott. 1988. Higher-order Abstract Syntax. *SIGPLAN Not.* 23, 7 (June 1988), 199–208. https://doi.org/10.1145/960116.54010

Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*. ACM, New York, NY, USA, Article 18, 13 pages. https://doi.org/10.1145/3236950.3236958

Cameron Swords and Daniel P. Friedman. 2013. rKanren: Guided Search in miniKanren. In *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming*.

Alfred Tarski. 1955. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific J. Math.* 5 (06 1955). https://doi.org/10.2140/pjm.1955.5.285

Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, Berlin, Heidelberg, 24–52. http://dl.acm.org/citation.cfm?id=647698.734146

Relational Processing for Fun and Diversity

Simulating a CPU relationally with miniKanren

GILMORE R. LUNDQUIST, University of Texas at Dallas, USA
UTSAV BHATT, University of Texas at Dallas, USA
KEVIN W. HAMLEN, University of Texas at Dallas, USA

Defining a central processing unit relationally using miniKanren is proposed as a new approach for realizing assembly code diversification. Software diversity has long been championed as a means of protecting digital ecosystems from widespread failures due to cyberattacks and faults, but is often difficult to achieve in practice. Using relational programming to simulate a processor allows large-scale automatic synthesis of assembly-level code. Early experiments with the technique indicate that such synthesis might lead to better automation of code diversification by breaking the synthesis problem into manageable chunks. An early prototype is presented, with some sample synthesis tasks and discussion of possible future applications.

CCS Concepts: • **Software and its engineering** → **Search-based software engineering**; *Software safety*; Interpreters; Source code generation; • **Security and privacy** → *Software security engineering*;

Additional Key Words and Phrases: program synthesis, relational programming, miniKanren, software artificial diversity, malware polymorphism

1 INTRODUCTION

Software diversity has long been recognized as valuable for protecting digital ecosystems from widespread failures due to cyberattacks and faults [Cohen 1993]. Higher diversity of software implementations reduces the likelihood that a single, common flaw pervades all its deployments, and therefore that a single attack can compromise all members of the ecosystem or affect them all in the same way. Unfortunately, most software ecosystems today remain highly monocultural—all deployments of a given software product for a given architecture are almost identical, save for minute differences. This homogeneity often allows individual, low-cost cyberattacks to have a devastating impact on large numbers of computer systems. For example, the 2015 Stagefright 2.0 vulnerability left nearly all Android devices susceptible to remote compromise due to almost identical multimedia library implementations being used by nearly all apps on all the devices [Peters 2015].

One reason why software monoculture continues to abound despite its brittleness against attack is the uniformity of most present-day tools for developing software products. Compilers are typically designed to solve an optimization problem that translates a given source program into a single, efficient, equivalently behaved object program. Compiler design goals therefore typically include semantic transparency (behavioral preservation), runtime efficiency, and space efficiency, but not diversity. The source semantics of mainstream imperative languages (e.g., C/C++), the optimization stages of their compilers, and their backend code generation algorithms are all designed to search for a single, good solution to this optimization problem.

Authors' addresses: Gilmore R. Lundquist, University of Texas at Dallas, USA, gilmore.lundquist@utdallas.edu; Utsav Bhatt, University of Texas at Dallas, USA, utsav.bhatt@utdallas.edu; Kevin W. Hamlen, University of Texas at Dallas, USA, hamlen@utdallas.edu.

2475-1421/2019/8-ART6

Our research seeks to shift the compiler optimization problem to pursue diversity as a design goal of software development. In particular, a diversifying software development methodology should attempt to yield a maximally dissimilar collection of object code implementations that are all semantically transparent to the original source code and that meet a certain baseline efficiency.

Although a few code diversification strategies have attained widespread deployment in practice, most provide only limited forms of diversity that continue to leave ecosystems vulnerable to exploitation. For example, *Address Space Layout Randomization* (ASLR) diversifies programs by randomly choosing the base addresses of libraries at load-time. While this does help mitigate some attacks, the resulting diversity is low, leaving the software vulnerable to *derandomization attacks* [Shacham et al. 2004]. We believe that changing implementations in a more fundamental way, such as modifying how values are computed, will protect against broader classes of attacks.

Historically, however, this type of diversity has been expensive to obtain and difficult to achieve in an automated fashion. Prior work [Lundquist et al. 2016] proposes merging the fields of artificial diversity of software with that of program synthesis, leveraging the natural diversity of search-style problems in order to create a plethora of program implementations. Because they spring from search problems, these implementations could be fundamentally different ways of solving a given computational problem.

One approach to program synthesis makes use of miniKanren[1] [Byrd 2009], a family of logic languages typically implemented as an embedded *Domain-Specific Language* (DSL). Since it is a relational language, miniKanren programmers write specifications that relate values, and users submit queries that yield sets of values within the relation. Since any value can queried by the system, values in computations that are traditionally thought of as "inputs" or "outputs" need not be. Querying for "inputs" that relate to a specified "output" effectively reverses the computation.

Prior work [Byrd et al. 2017, 2012] has proposed realizing program synthesis in miniKanren by implementing the relational specification of an interpreter that relates input code to the output values it produces. Reversing the computation by querying for inputs from given outputs then produces possible code that produces the desired output values.

Program synthesis in miniKanren seems particularly well suited to the goal of code diversity given miniKanren's natural ability to easily produce multiple answers for a query. The user simply specifies how many answers (at a maximum) the system is to search for, and a list of results is returned.

In the interest of being source code-agnostic with our diversification efforts, we would like to synthesize programs at the assembly level. This potentially has the advantages of breaking the synthesis problem into manageable chunks, as well as allowing existing code (of any origin) to be diversified in a largely automated fashion.

Since we wish to achieve program diversity at the assembly level, we propose writing a relational specification of a central processing unit. This miniKanren program interprets assembly code by relating that code to the states of the processor before and after code execution. A user can then query for assembly code that produces a desired output state.

2 MODES OF OPERATION

We envision the following ways of using an assembly language synthesis system, such as our miniKanren prototype:

2.1 For Diversification

Our main goal is to explore automated diversification of assembly-level code. Potential approaches include the following:

[1] See http://minikanren.org

- Use of code sketches:
  A standard approach to program synthesis is that of the *program sketch* [Solar-Lezama 2008, 2009]: a partial program with holes to be filled with synthesized code. In a logic programming environment, a sketch is a partial program with portions (here instruction mnemonics, instruction arguments, etc.) represented by logic variables. Those variables are then queried using a `run` command to determine possible values. To control which output is being computed, a sketch must also include a formal specification of what is to be computed. In assembly diversification, these specifications are constraints dictating the possible processor states after a computation runs. (We use sketches in this paper.)
  Constraints collectively define correctness of synthesized answers. Correct synthesis algorithms yield only answers that are correct with respect to the constraints. For code diversification, constraints must therefore also define what is meant by program equivalence. For example, a user may decide to specify result values for processor status flags, or leave them unconstrained. The latter option broadens the definition of equivalence to include relations on processor states. For example, diversification might be required to preserve register and memory values but be permitted to vary status flag values.
  When diversifying existing code, either the program to be diversified already has a formal (mathematical) specification, or (more commonly) there will *a priori* be no formal notion of correctness. In the ideal former case, we need only convert the specification into miniKanren constraints. In the latter case, quality assurance and testing processes must be applied to check program correctness or equivalence; however, for diversity to be tractable, this assurance process should be (semi-)automated to easily apply it to the multitude of variants generated.
- Basic blocks bracketed by control flow instructions:
  Generating proper control-flow using program synthesis is known to be a hard problem [Solar-Lezama 2008]. To avoid this, sketches can explicitly specify most or all control-flow transfer instructions [Solar-Lezama 2008]. Instructions in the basic block(s) between control-flow transfers are synthesized in entirety or in part. Program constraints include loop invariants to dictate what is synthesized. (The GCD examples in the following section illustrate this approach.)
- Use of effect traces:
  In addition to dividing code up into basic blocks, another approach is to generate code based on a known effect trace, synthesizing portions between calls to other functions, system routines, or other side effects not otherwise modeled by our processor relation. The format and construction of arguments to these external routines are determined by constraints provided in the sketch. Since trace equivalence is a common way to define program equivalence, this approach provides an intuitive starting point for solving the equivalence problem mentioned above.
- Diversification of existing code:
  One of our primary goals for this project is the automated diversification of existing code. To achieve this, processor states could be generated by running existing assembly code forwards in our system. Enhanced by constraints describing known properties of how the code should function, these processor states become the end goals for new code to satisfy.
- Gadget-oriented program composition:
  A *gadget* can be defined as any arbitrary string of assembly language instructions, usually one found in pre-existing code. As shown in prior work [Lundquist et al. 2016; Mohan and Hamlen 2012] (and as shown by Return-Oriented Programming in general [Schwartz et al. 2011]), programs can be constructed by stringing chains of gadgets together in an order determined by the synthesis engine as a means of reaching a particular goal. Return-Oriented Programming (ROP) is an exploit technique that repurposes gadgets found in existing benign code to implement attack payloads. While gadget-oriented programming was

originally used for malicious purposes, we speculate that this technique could be adapted for more general synthesis tasks.

The overall theme of the techniques described above is that of breaking the problem of synthesis into small, manageable chunks at a low level. Rather than trying to synthesize an entire program at once, a program is broken down into sub-pieces, which are then synthesized. This avoids traditional problems associated with synthesizing large or complex code fragments, and mitigates state-space explosion.

2.2 For Use by a Compiler

Since programming using processor states and constraint sets is likely to be difficult, we envision this system as a target of higher level tools, such as compilers or interpreters. Replacing the back-end code generation of a compiler with a synthesis system allows for automatic diversification while keeping programming tasks manageable. Compilers are potentially in a unique position to know what constraints must be preserved from higher-level source in the generated assembly. Further, allowing compilers and other code-generation tools to programmatically break the result into known intermediate states allows for smaller and more frequent synthesis tasks, once again keeping with the theme of reducing synthesis problems into smaller sub-problems. Sub-problems can be divided at whatever level is appropriate for the tool based on its knowledge and analysis of the code being generated, or based on what size tasks the synthesis engine is capable of handling efficiently.

We plan to use our system to explore each of the above approaches in future work.

3 IMPLEMENTATION

To explore assembly language synthesis, we have developed a prototype relational assembly interpreter in miniKanren.[2] Our prototype uses a subset of Intel x86 assembly language instructions with the limited set of 32-bit general purpose registers available for that architecture. The implementation uses a modified version of faster-miniKanren[3] written in Racket.

A full scale system must support enough instructions and processor features to synthesize programs that work; but like existing compiler back-ends (which emit only a subset of the available instructions), this does not necessitate supporting the entire instruction set architecture (ISA). Supporting more of the ISA allows for more diversity in the resulting programs, potentially at the cost of synthesis time. However, prior work [Mohan and Hamlen 2012] has found that only a small portion of the architecture is needed to achieve high code diversity.

3.1 The x86^o Relation

Our processor relation `x86`^{o} relates assembly code, an input processor state, and an output processor state. Processor state is modeled as a set of association lists, one mapping registers to values and another mapping memory locations to values. The memory mapping is a partial function, only containing those values that have been updated by the program. Addresses read from an uninitialized address return a default value, typically `0`.

The interpreter models assembly code as a list of instructions, using a fall-through approach to execute (or synthesize) a basic block. Since we do not yet represent code addresses in our model, execution always necessarily continues to the subsequent instruction in the list. Each instruction is a list containing an instruction mnemonic and the appropriate number of operands for the instruction. Each operand describes a register, an immediate value, or a memory address.

[2]Code is available for download at https://www.utdallas.edu/~hamlen/lundquist-miniKanren19.zip or https://www.utdallas.edu/~hamlen/lundquist-miniKanren19.tar.gz.

[3]Obtained from https://github.com/gregr/tutorial-relational-interpreters. This version contains modifications by Greg Rosenblatt to speed up the `eval`^{o} relational interpreter. The original faster-miniKanren can be obtained from https://github.com/michaelballantyne/faster-miniKanren/blob/master/README.md.

The interpreter calls sub-relations to decode arguments (which relate operands to their values with respect to some processor state) and then interprets the instruction by operating on those values and creating an updated processor state. A fragment of the relation with a few instructions is shown below:

```
;; The x86 processor relation.
(define (x86$^o$ code rstore new-rstore)
  (conde
    [(≡ '() code) (≡ rstore new-rstore)]

    [(fresh (opcode arglist morecode op1 op2 res rstore1)
       (≡ code  ((,opcode . ,arglist) . ,morecode))
       (conde

         ;; add
         [(≡ opcode 'add)
          (decode-2args$^o$ arglist op1 op2 rstore)
          (plus$^o$ op1 op2 res)
          (update-arg$^o$ arglist res rstore rstore1)
          (x86$^o$ morecode rstore1 new-rstore)]
...
         ;; div
         ;;  uses implicit arguments -
         ;;    dividend is edx:eax,
         ;;    divisor is op1,
         ;;    destination (result quotient) is eax,
         ;;    (result) remainder is edx
         [(≡ opcode 'div)
          (decode-1arg$^o$ arglist op1 rstore)
          (fresh (edx eax n rem rstore2)
            (lookup$^o$ 'R_EDX rstore eax)
            (lookup$^o$ 'R_EAX rstore edx)
            (append$^o$ eax edx n) ; n=dividend: eax=low order bits, edx=high order bits
            (div$^o$ n op1 res rem)
            (update$^o$ 'R_EAX res rstore rstore1)
            (update$^o$ 'R_EDX rem rstore1 rstore2)
            (x86$^o$ morecode rstore2 new-rstore))]
...
         ;; xor
         [(≡ opcode 'xor)
          (decode-2args$^o$ arglist op1 op2 rstore)
          (xor$^o$ op1 op2 res)
          (update-arg$^o$ arglist res rstore rstore1)
          (x86$^o$ morecode rstore1 new-rstore)]
...
```

Numeric values in our system are represented as *Oleg numerals*—little-endian lists of binary digits that encode the base-2 representation of the number. Most arithmetic operations are the definitions[4] found in *The Reasoned Schemer* [Friedman et al. 2018], with a few of our own added to implement missing operators (e.g., logical operations) needed for the instructions we've included. For example, here is our implementation of integer division and remainder:

```
; integer division with remainder
(define (div^o a b q r)
  (conde
    [(≡ q '()) (≡ r a) (<l^o a b)]
    [(fresh (p)
       (<=l^o b a)
       (<l^o r b)
       (plus^o p r a)
       (*^o b q p))]))
```

3.2 Example Queries

3.2.1 Synthesis 1: Generating Some Assembly. Our first example interaction demonstrates a query for generating lots of assembly code quickly. The goal of each synthesized program is to place the result value 1875 (Oleg numeral (1 1 0 0 1 0 1 0 1 1 1)) into register EAX.

We begin by providing the following assembly program sketch:

```
mov ECX ?x
mov EDX ?y
mov EAX ?p
mov EBX ?q
mov EDI ?s
mov ESI ?t

?g ECX ESI
?f EDX EDI
?d EAX ECX
?e EBX EDX
?c EAX EBX
```

The question marks denote holes to be filled in by the synthesizer, subscripted with the name of the logic variable used to denote the hole. The first six instructions denote `mov` instructions with a particular destination operand (one for each general-purpose register) and a hole to be filled in for the source operand. The remaining five instructions give concrete destination and source operands but leave the choice of instruction open to be synthesized.

Constraints are then added to further limit what will be synthesized. The first set of constraints prevent any of the final instructions from being additional `mov` instructions. The second set prevents some source operands from

[4]Obtained from https://github.com/miniKanren/CodeFromTheReasonedSchemer2ndEd.

being the same as some of the other source operands. The third set of constraints disallow source operand holes from containing 1875, which prevents the goal value from being moved directly into a register. Finally, source operands must be positive (non-zero).

We use our processor relation to relate the sketch code, an initial empty processor state, and a final processor state. Constraining this final state by the value in EAX forces our desired goal to be met. We then query for the fully synthesized code (in variable v1) and the resulting processor state (in variable z).

For our queries, we make use of Racket's `time` operator, which reports (in milliseconds) CPU time, real time, and garbage collection time spent for expression evaluation. The full query and first 2 (of 200) results are as follows:

```
> (time
    (run 200 (v1 z)
      (fresh (d x p e q y c s t f g)
         (≡ v1  ((mov R_ECX ,x)
                 (mov R_EDX ,y)
                 (mov R_EAX ,p)
                 (mov R_EBX ,q)
                 (mov R_EDI ,s)
                 (mov R_ESI ,t)

                 (,g R_ECX R_ESI)
                 (,f R_EDX R_EDI)
                 (,d R_EAX R_ECX)
                 (,e R_EBX R_EDX)
                 (,c R_EAX R_EBX)))
         (≠ c 'mov) (≠ d 'mov) (≠ e 'mov) (≠ f 'mov) (≠ g 'mov)
         (≠ p q) (≠ x q) (≠ y p) (≠ y q) (≠ x p) (≠ x y)
         (≠ x '(1 1 0 0 1 0 1 0 1 1 1))
         (≠ y '(1 1 0 0 1 0 1 0 1 1 1))
         (≠ q '(1 1 0 0 1 0 1 0 1 1 1))
         (≠ p '(1 1 0 0 1 0 1 0 1 1 1))
         (≠ s '(1 1 0 0 1 0 1 0 1 1 1))
         (≠ t '(1 1 0 0 1 0 1 0 1 1 1))
         (pos^o x) (pos^o y) (pos^o p) (pos^o q) (pos^o t) (pos^o s)
         (x86^o v1 initial-store z)
         (lookup^o 'R_EAX z '(1 1 0 0 1 0 1 0 1 1 1))
  )))
cpu time: 21109 real time: 21481 gc time: 9204
'(((((mov R_ECX (_.0 . _.1))
     (mov R_EDX (_.2 . _.3))
     (mov R_EAX (1))
     (mov R_EBX (0 1 0 0 1 0 1 0 1 1 1))
     (mov R_EDI (_.2 . _.3))
     (mov R_ESI (_.0 . _.1))
     (sub R_ECX R_ESI)
     (sub R_EDX R_EDI)
```

```
     (add R_EAX R_ECX)
     (add R_EBX R_EDX)
     (add R_EAX R_EBX))
    ((R_EAX 1 1 0 0 1 0 1 0 1 1 1) (R_EBX 0 1 0 0 1 0 1 0 1 1 1) (R_ECX) (R_EDX)
     (R_EDI _.2 . _.3) (R_ESI _.0 . _.1)))
   (=/=
    ((_.0 0) (_.1 (1 0 0 1 0 1 0 1 1 1)))
    ((_.0 1) (_.1 (1 0 0 1 0 1 0 1 1 1)))
    ((_.0 1) (_.1 ()))
    ((_.0 _.2) (_.1 _.3))
    ((_.2 0) (_.3 (1 0 0 1 0 1 0 1 1 1)))
    ((_.2 1) (_.3 (1 0 0 1 0 1 0 1 1 1)))
    ((_.2 1) (_.3 ())))))
  ((((mov R_ECX (_.0 . _.1))
     (mov R_EDX (_.2 . _.3))
     (mov R_EAX (0 1 0 0 1 0 1 0 1 1 1))
     (mov R_EBX (1))
     (mov R_EDI (_.2 . _.3))
     (mov R_ESI (_.0 . _.1))
     (sub R_ECX R_ESI)
     (sub R_EDX R_EDI)
     (add R_EAX R_ECX)
     (add R_EBX R_EDX)
     (add R_EAX R_EBX))
   ((R_EAX 1 1 0 0 1 0 1 0 1 1 1) (R_EBX 1) (R_ECX) (R_EDX) (R_EDI _.2 . _.3) (R_ESI _.0 . _.1)))
   (=/=
    ((_.0 0) (_.1 (1 0 0 1 0 1 0 1 1 1)))
    ((_.0 1) (_.1 (1 0 0 1 0 1 0 1 1 1)))
    ((_.0 1) (_.1 ()))
    ((_.0 _.2) (_.1 _.3))
    ((_.2 0) (_.3 (1 0 0 1 0 1 0 1 1 1)))
    ((_.2 1) (_.3 (1 0 0 1 0 1 0 1 1 1)))
    ((_.2 1) (_.3 ()))))
```

...

3.2.2 Synthesis 2: Reverse ALU. Next we give an example of *Angelic Execution*—determining which sets of inputs successfully result in a particular output [Bodik et al. 2010; Chandra et al. 2011]. Here the output value 65,535 must be assigned to register EAX after computing the following assembly fragment:

```
mul EAX EBX
or ECX EDX
add ESI EDI
dec EAX
xor ESI ECX
inc ECX
xor EAX ECX
```

Each synthesized output contains a set of possible positive input values, one for each of the six general-purpose registers. Each input set successfully results in the desired output.

The full query and first 5 (of 200) results are shown below:

```
> (time
    (run 200 (v1 v2 v3 v4 v5 v6)
      (fresh (a b c x y z str)
        (pos^o a) (pos^o b) (pos^o c) (pos^o x) (pos^o y) (pos^o z)
        (x86^o  ((mov R_EAX ,x)
                 (mov R_EBX ,y)
                 (mov R_ECX ,z)
                 (mov R_EDX ,a)
                 (mov R_ESI ,b)
                 (mov R_EDI ,c)
                 (mul R_EAX R_EBX)
                 (or R_ECX R_EDX)
                 (add R_ESI R_EDI)
                 (dec R_EAX)
                 (xor R_ESI R_ECX)
                 (inc R_ECX)
                 (xor R_EAX R_ECX))
               initial-store
               str)
        (lookup^o 'R_EAX str '(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1))
        (≡  (R_EAX ,x) v1)
        (≡  (R_EBX ,y) v2)
        (≡  (R_ECX ,z) v3)
        (≡  (R_EDX ,a) v4)
        (≡  (R_ESI ,b) v5)
        (≡  (R_EDI ,c) v6)
  )))
cpu time: 14062 real time: 14129 gc time: 4384
'(((R_EAX (0 1 1 1 1 1 1 1 1 1 1 1 1 1 1)) (R_EBX (1)) (R_ECX (1))
   (R_EDX (1)) (R_ESI (1)) (R_EDI (1)))
  ((R_EAX (1)) (R_EBX (0 1 1 1 1 1 1 1 1 1 1 1 1 1 1)) (R_ECX (1))
   (R_EDX (1)) (R_ESI (1)) (R_EDI (1)))
  ((R_EAX (0 1 1 1 1 1 1 1 1 1 1 1 1 1 1)) (R_EBX (1)) (R_ECX (1))
   (R_EDX (1)) (R_ESI (1)) (R_EDI (0 _.0 . _.1)))
  ((R_EAX (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)) (R_EBX (0 1)) (R_ECX (1))
   (R_EDX (1)) (R_ESI (1)) (R_EDI (1)))
  ((R_EAX (0 1)) (R_EBX (1)) (R_ECX (1))
   (R_EDX (0 0 1 1 1 1 1 1 1 1 1 1 1 1 1)) (R_ESI (1)) (R_EDI (1)))

...
```

3.2.3 Specifying Loop Constraints: Euclid's GCD Algorithm. For our final example, we use a slightly higher-level code fragment to demonstrate how we handle looping or stitching together multiple code blocks. Our prototype currently only handles straight-line code with no jumps. To specify code with loops or more complicated control flow, one can specify how the blocks are to be sequenced using miniKanren.

As our example, we use Euclid's algorithm for finding the Greatest Common Divisor (GCD) of two integers a and b. Recall that the algorithm proceeds as follows:

(1) Find the remainder r when dividing a and b. (Equivalently, find integers q and r such that $a = qb + r$).
(2) If $r = 0$ then b already divides both a and b, and is the largest integer that divides both a and b; return b.
(3) Otherwise, any number that divides a and b must also divide b and r. Repeat from step (1) to find the GCD of b and r.

To implement this algorithm, first we adopt the convention that the values for a and b are held in registers EAX and EBX, respectively. We then write a miniKanren relation `gcd_tester`^{o} to run the loop. Given an assembly code body and input values, the code is run in a processor environment with values in their proper registers. It then checks the remainder result, and either associates the output with the final value (if the remainder is zero) or recursively calls itself on the resulting b and r values.

```
; recursive loop which calls the basic block [the loop body]
(define (gcd_tester^o code a b gcd)
  (fresh (s s+ stmp)
    (update^o 'R_EAX a initial-store stmp)
    (update^o 'R_EBX b stmp s)
    (gcd_block_code^o code s s+)
    (conde
      [(lookup^o 'R_EBX s+ '() )  ; final remainder is 0
       (lookup^o 'R_EAX s+ gcd)]  ; result is in EAX
      [(fresh (r)
         (lookup^o 'R_EBX s+ r)
         (≠ r '()) ; r is non-0
         (gcd_tester^o code b r gcd))])))  ; otherwise loop
```

Relation `gcd_block_code`^{o} specifies a loop invariant for our algorithm. It runs our x86 simulator on the given code, assuming the above register convention and specifying mathematical constraints for outputs that a correct run should produce. In this case, the constraints require that after the code runs, the new a value is the original b value and the new b value is the remainder of a and b.

```
; spec for gcd loop invariant
(define (gcd_block_code^o code s s+)
  (fresh (old_eax old_ebx new_eax new_ebx q r)
    (lookup^o 'R_EAX s old_eax)  ; a
    (lookup^o 'R_EBX s old_ebx)  ; b
    (x86^o code s s+)
    (lookup^o 'R_EAX s+ new_eax)
    (lookup^o 'R_EBX s+ new_ebx)
    (≡ new_eax old_ebx)  ; a ← b
    (div^o old_eax old_ebx q r)  ; a = qb + r,  r < b
    (≡ new_ebx r)))
```

Using this infrastructure, we now run an algorithm on our processor and verify the answers computed are correct.

```
; euclid's gcd algorithm
(define euclid '(
  (xor R_EDX R_EDX)
  (div R_EBX)
  (mov R_EAX R_EBX)
  (mov  R_EBX R_EDX)))

; test euclid's algorithm and get an answer
> (time (run* (d) (gcd_tester^o euclid (build-num 12) (build-num 9) d)))
cpu time: 78 real time: 80 gc time: 0
'((1 1))

> (time (run* (d) (gcd_tester^o euclid (build-num 42) (build-num 30) d)))
cpu time: 203 real time: 213 gc time: 15
'((0 1 1))
```

We see that we get the correct answers of 3 and 6, resepectively (in Oleg numeral representation). As with any miniKanren relation, we can query for multiple answers at once; here we simultaneously obtain the gcd of 12 and all values of b up to 12, and verify that the answers are correct:

```
> (time (run* (b d) (<=^o b (build-num 12)) (gcd_tester^o euclid (build-num 12) b d)))
cpu time: 1329 real time: 1336 gc time: 937
'(((0 0 1 1) (0 0 1 1))
  ((1) (1))
  ((0 1) (0 1))
  ((1 1) (1 1))
  ((0 0 1) (0 0 1))
  ((0 1 1) (0 1 1))
  ((1 1 0 1) (1))
  ((1 0 1) (1))
  ((0 1 0 1) (0 1))
  ((0 0 0 1) (0 0 1))
  ((1 0 0 1) (1 1)))
```

Interestingly, we see three different classes of answers to this query, generated in order: the first six are those for which $b = d$, namely the factors of 12 (12, 1, 2, 3, 4, and 6). The next group (11 and 5) are those that are *relatively prime* with 12, for which the only common divisor is 1. The remaining three answers (10, 8, and 9) are those that are not factors of 12, yet still have common factors (2, 4, and 3, respectively) with 12.

3.2.4 Synthesis 3: Synthesizing Euclid's Algorithm with a Sketch. We now synthesize a similar algorithm by querying for the code. An initial, naïve approach attempts to synthesize the code directly using the relations we have so far:

```
> (time (run 1 (c) (gcd_tester^o c (build-num 12) (build-num 9) (build-num 3))))
. . .
    ; [Fails with out-of-memory error]
```

The out of memory error occurs because this naïve query is too vague, resulting in a state space explosion. To narrow the search space, we employ a program sketch. Using the intuition that only a division operation and moving some data around should be sufficient, we create a program sketch that only allows `div` and `mov` instructions, with a specified program length. We take advantage of relational programming to auto-generate a suitable sketch:

```
(define (gen-mov-div-listᵒ depth out)
  (conde
    [(≡ depth '()) (≡ out '())]
    [(≠ depth '())
     (fresh (n l)
        (minusᵒ depth '(1) n)
        (conde
          [(fresh (op1 op2) (≡ out  ((mov ,op1 ,op2) . ,l) ))]
          [(fresh (op)      (≡ out  ((div ,op)       . ,l) ))]
        )
        (gen-mov-div-listᵒ n l) )]))

> (time (run 1 (c) (gen-mov-div-listᵒ (build-num 4) c)
                   (gcd_testerᵒ c (build-num 12) (build-num 9) (build-num 3))))
cpu time: 361204 real time: 362308 gc time: 84412
'(((mov R_ECX ()) (div R_EBX) (mov R_EAX R_EBX) (mov R_EBX R_EDX)))
```

Here we have limited possible sketches to programs of length 4 with only `div` or `mov` instructions, and the system is now able to produce an answer. Upon running the test queries below, we see that this new code produces the same correct GCD answers as the original code above.

```
(define new-code '((mov R_ECX ()) (div R_EBX) (mov R_EAX R_EBX) (mov R_EBX R_EDX)))
> (time (run* (d) (gcd_testerᵒ new-code (build-num 12) (build-num 9) d)))
cpu time: 110 real time: 107 gc time: 46
'((1 1))
> (time (run* (d) (gcd_testerᵒ new-code (build-num 42) (build-num 30) d)))
cpu time: 265 real time: 262 gc time: 31
'((0 1 1))
> (time (run* (b d) (<=ᵒ b (build-num 12)) (gcd_testerᵒ new-code (build-num 12) b d)))
cpu time: 422 real time: 437 gc time: 76
'(((0 0 1 1) (0 0 1 1))
  ((1) (1))
  ((0 1) (0 1))
  ((1 1) (1 1))
  ((0 0 1) (0 0 1))
  ((0 1 1) (0 1 1))
  ((1 1 0 1) (1))
  ((1 0 1) (1))
  ((0 1 0 1) (0 1))
  ((0 0 0 1) (0 0 1))
  ((1 0 0 1) (1 1)))
```

4 RELATED WORK

Rosette [Torlak and Bodik 2013] is another DSL embedded in Racket capable of program synthesis and angelic execution. Instead of backtracking search, Rosette uses a *satisfiability modulo theories* (SMT) solver to find solutions to synthesis and constraint problems. This gives it the potential to be more efficient than miniKanren in finding solutions that are arithmetic in nature. While any sufficiently general program synthesis system should be able to synthesize assembly code in a similar fashion to our approach, we found the Rosette system to be focused on returning a single optimal answer to queries. This makes it more cumbersome to achieve large-scale diversity with Rosette than with miniKanren in our experience.

Minimips[5] is a miniKanren implementation of a MIPS architecture assembler/dissassembler. Its relation converts between MIPS assembly language programs and their binary encodings. Minimips contains a full syntactic description for MIPS instructions, but doesn't appear to have an interpreter. It therefore has no semantic description of the instructions, or the ability to synthesize code that satisfies a formal specification. We chose to use the x86 architecture rather than a RISC architecture such as MIPS because a CISC architecture naturally allows for more diversity of implementations. As future work we plan to implement a similar relational assembler/disassembler for our x86 system.

Automated generation of assembly code is a common task of compilers and other similar tools. As noted in Section 1, these tools do not have diversity of implementation as a goal. Some systems (e.g., [Hong and Gerber 1993; Pu et al. 1988]) do synthesize assembly code for specific tasks using algorithms unique to the task. However, these systems do not synthesize general-purpose assembly code from arbitrary program constraints, or for the purpose of implementation diversity.

5 CONCLUSION

This paper proposed the implementation of an assembly-level interpreter in miniKanren for the purpose of synthesizing assembly code, motivated by the need for increased software diversity. This allows synthesis problems to be broken up into small, manageable pieces. Such problems can be subdivided using specific sketches, basic blocks, or effect traces; and can be driven by various inputs, including effect traces, processor states obtained from existing code, availability of particular gadgets, or compiler-driven information. Our working prototype implements an assembly interpreter for a small subset of x86, and experiments demonstrate its use for synthesizing code in both straight-line and looping programs. Synthesis examples show the potential to synthesize large numbers of diverse implementations.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and William Byrd for their helpful comments on this paper. We would also like to thank the committee chair Nada Amin and all of the miniKanren workshop organizers. The research reported herein was supported in part by NSF Award #1513704, ONR Award N00014-17-1-2995, and an endowment from the Eugene McDermott family. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the above supporters.

REFERENCES

Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with Angelic Nondeterminism. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 339–352.

William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University, Bloomington, Indiana.

[5] https://github.com/orchid-hybrid/minimips

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 (PACMPL)* 1 (2017), 8:1–8:26.

William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the Annual Workshop on Scheme and Functional Programming*. 8–29.

Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. 121–130.

Fred B. Cohen. 1993. Operating System Protection Through Program Evolution. *Computers & Security* 12, 6 (1993), 565–584.

Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer* (second ed.). MIT Press.

Seongsoo Hong and Richard Gerber. 1993. Compiling Real-time Programs into Schedulable Code. In *Proceedings of the 14th ACM Conference on Programming Language Design and Implementation (PLDI)*, Vol. 28. 166–176.

Gilmore R. Lundquist, Vishwath Mohan, and Kevin W. Hamlen. 2016. Searching for Software Diversity: Attaining Artificial Diversity through Program Synthesis. In *Proceedings of the New Security Paradigms Workshop (NSPW)*. 80–91.

Vishwath Mohan and Kevin W. Hamlen. 2012. Frankenstein: Stitching Malware From Benign Binaries. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*. 77–84.

Sara Peters. 2015. Stagefright 2.0 Vuln Affects Nearly All Android Devices. *DARKReading* (October 2015).

Calton Pu, Henry Massalin, and John Ioannidis. 1988. The Synthesis Kernel. *Computing Systems* 1, 1 (1988), 11–32.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Security Symposium*. 25–41.

Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. 298–307.

Armando Solar-Lezama. 2008. *Program Synthesis By Sketching*. Ph.D. Dissertation. The University of California, Berkeley.

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS)*. 4–13.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Proceedings of the 3rd ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)* 135–152.