An Annotated Implementation of miniKanren with Constraints

Bharathi Ramana Joshi
IIIT Hyderabad
India

William E. Byrd
University of Alabama at Birmingham
USA

Abstract

We show how to implement miniKanren with `=/=`, `symbolo`, `numbero`, and `absento`. The set of constraints we choose is motivated by the fact that interesting classes programs can be implemented with these, e.g. relational interpreters and relational type inferencers. We also present a four-step constraint implementation recipe that enables readers to implement constraints beyond the set we chose.

Keywords: miniKanren implementation, Racket, relational programming

ACM Reference Format:
Bharathi Ramana Joshi and William E. Byrd. 20022. An Annotated Implementation of miniKanren with Constraints. In *miniKanren '22, International Conference on Functional Programming, September 11–16, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Many interesting relational programs such as relational interpreters and relational type inferencers require certain relational constraints (`=/=`, `numbero`, `symbolo`, and `absento`). Therefore, if one wants to learn how to implement these constraints, one is compelled to study existing implementations. However, the two well-known implementations — faster-miniKanren [Ballantyne 2021] and Appendix D from Byrd et al. [2012] — make accomplishing this goal difficult for different reasons. The former is highly optimized for performance, and thus is not ideal to learn the ideas underlying implementing these constraints. The latter appendix is orthogonal to the problem solved by the paper and is not written with the intent to teach constraint implementation. As a result, it is hard to learn constraint implementation from this appendix. Thus, one interested in learning to implement miniKanren with constraints lacks an ideal resource.

This paper aims to demonstrate how to implement miniKanren with constraints. We walk the reader through implementing miniKanren with constraints capable of running the relational interpreter for quine generation [Byrd et al. 2012]. In particular, by the end of this paper, the reader will have implemented:

1. the `=/=` constraint;
2. `numbero` and `symbolo` type constraints;
3. the `absento` constraint;
4. and reification in the presence of above constraints.

microKanren [Hemann and Friedman 2013] serves as an excellent means to learn how to implement a minimalistic purely relational programming language. Thus, we take microKanren as our starting point and incrementally add the above constraints to it.

1.1 Prerequisites

Firstly, proficiency with programming in Racket is required as this tutorial builds up an implementation of miniKanren in Racket. However, Scheme programmers will also be able to follow the paper with no difficulty. Next, proficiency with relational programming in miniKanren is also assumed. In particular, we assume the reader knows how to use the constraints we implement. We direct unfamiliar readers to Byrd et al. [2017] for the same. Finally, we assume the reader has implemented microKanren [Hemann and Friedman 2013] and implement our miniKanren on top of this.

2 Implementation

We start with microKanren [Hemann and Friedman 2013], and incrementally add constraints to it. Thus, we assume the reader has implemented `==`, `fresh`, conjunction, and disjunction.

The syntax of runnable programs to begin with is:

$$
\begin{array}{lcl}
\langle \textit{program} \rangle & ::= & (\texttt{run}\ \langle \textit{number} \rangle\ (\langle \textit{id} \rangle)\ \langle \textit{goal-expr} \rangle) \\
 & | & (\texttt{run*}\ (\langle \textit{id} \rangle)\ \langle \textit{goal-expr} \rangle) \\
\langle \textit{goal-expr} \rangle & ::= & (\texttt{==}\ \langle \textit{term-expr} \rangle\ \langle \textit{term-expr} \rangle) \\
 & | & (\texttt{fresh}\ (\langle \textit{id} \rangle+)\ \langle \textit{goal-expr} \rangle+) \\
 & | & (\texttt{conde}\ (\langle \textit{goal-expr} \rangle+)+)
\end{array}
$$

```
⟨term-expr⟩ ::= (quote ⟨value⟩)
              | ⟨id⟩
              | ⟨value⟩
              | (cons ⟨term-expr⟩ ⟨term-expr⟩)
```

⟨*value*⟩ ::= A Racket number or symbol or the empty list

⟨*id*⟩ ::= Any valid Racket identifier

Term expressions evaluate to terms, whose grammar is:

```
⟨term⟩ ::= ⟨logic-var⟩
         | ⟨symbol⟩
         | ⟨number⟩
         | ⟨pair⟩
```

⟨*logic-var*⟩ ::= miniKanren logic variable

⟨*symbol*⟩ ::= Any valid Racket symbol

⟨*number*⟩ ::= Any valid Racket number

⟨*pair*⟩ ::= Any valid Racket pair of ⟨*term*⟩s

One major change we make to the microKanren implementation is using a list for the state instead of a pair. In microKanren, the state consisted of two components — the substitution and the fresh variable counter. Thus, implementing the state as a pair was sufficient. However, as we implement constraints we will have to extend the state. Therefore, we use a list for the state and add new elements to the list to extend the state.

Before proceeding further, it is useful to outline the recipe for implementing any constraint (including constraints not discussed in this paper). This equips the reader with the essential idea of implementing constraints, which is far more valuable than learning to implement the limited set of constraints we discuss in this paper. The recipe to implement a new constraint `c` is:

1. extend the state with a field to store the information required by `c`;
2. define how `c` updates the state when invoked;
3. perform an exhaustive case-wise analysis on how `c` interacts with other constraints, and update both `c` and other constraints appropriately; and
4. update the reifier to display required information from the extended state.

For each of the constraints we implement now, we follow the above recipe.

2.1 Disequality

In this section, we extend the implementation to handle the `=/=` constraint. Thus, the high level syntax of runnable programs is:

```
⟨goal-expr⟩ ::= (== ⟨term-expr⟩ ⟨term-expr⟩)
              | (fresh (⟨id⟩+) ⟨goal-expr⟩+)
              | (conde (⟨goal-expr⟩+)+)
              | (=/= ⟨term-expr⟩ ⟨term-expr⟩)
```

Rest of the grammar remains the same.

Using our recipe, implementing `=/=` can be broken down into four steps:

1. extend the state to include a disequality constraint store;
2. define the `=/=` goal constructor to update the disequality constraint store;
3. verify after each unification that no disequality constraints are violated, and possibly simplify the disequality constraint store; and
4. update the reifier to display information from the disequality store.

2.1.1 Extending the State.

First we have extend the state to hold a disequality store in addition to a substitution and a counter.

```
(define (make-st S C D)
  `(,S ,C ,D))

(define (S-of st)
  (car st))

(define (C-of st)
  (cadr st))

(define (D-of st)
  (caddr st))

(define empty-state (make-st '() 0 '()))
```

The disequality store is a list of association lists, where each association list contains information corresponding to a single disequality constraint. To understand what an association list in the store means, consider the following fragment of a program:

```
(fresh (x y)
  (=/= `(,x 3) `(cat ,y)))
```

This `=/=` constraint is violated only when the logic variable `x` is unified with the symbol `cat` *and* the logic variable `y` is unified with `3`. In other words, a disequality constraint is a conjunction of disequalities between ***atomic terms*** (booleans, symbols, numbers, the empty list, and logic variables). Specifically, each disequality must involve at least one fresh logic variable (since disequalities between constants are immediately resolved). We use the phrase ***atomic disequality*** to refer to disequalities between two atomic terms, at least one of which is a fresh logic variable. Thus, every disequality constraint is a conjunction of atomic disequalities. For the above example, the disequality constraint consists of the two atomic disequalities `(=/= x cat)` and `(=/= y 3)`. This can be captured by the following association list:

```
((y . 3) (x . cat))
```

That is, each pair in the association list corresponds to a single atomic disequality, and thus a disequality constraint is violated if all of its atomic disequalities are violated in conjunction. In the above example, the disequality constraint would be violated if the logic variable `y` were unified with `3` and the logic variable `x` with the symbol `cat`.

As a more complicated example, consider the following program fragment:

```
(fresh (x y)
  (=/= `(,x apple) `(banana ,y))
  (=/= `(,x 5) `(7 ,y)))
```

The corresponding disequality store would be:

```
  (((y . 5) (x . 7))
   ((y . apple) (x . banana))))
```

2.1.2 Defining the constraint. Next, we have to define `=/=` to update the disequality store. We reuse `unify` to define `=/=`. Consider `=/=` applied to terms say `t1` and `t2`. When we try to unify `t1` and `t2` in the current substitution `S`, three cases are possible.

1. `unify` fails. In this case, there is no possible way of extending `S` to make `(== t1 t2)` hold (if there was, `unify` would have returned this extended substitution). Thus, the disequality constraint can safely be discarded, as it already holds.
2. `unify` succeeds without extending `S`. In this case, `t1` and `t2` are already equal, meaning the `=/=` constraint is violated.
3. `unify` succeeds and returns an extended substitution. In this case, the extension of the returned substitution with respect to `S` contains precisely all those equalities between atomic terms that must hold for the unification to succeed. In other words, for the `=/=` constraint to hold, the extension must not hold! Thus, we add the extension to our disequality store as the new disequality constraint.

Examples for each of the above cases are as follows:

1. `(=/= 1 2)` may be discarded safely as unification of 1 and 2 fails.
2. `(=/= 1 1)` violates the `=/=` constraint as unification of 1 and 1 succeeds in any substitution `S` without extending `S`.
3. The previous example:
   `(fresh (x y) (=/= `(,x 3) `(cat ,y)))`
   is a case where we have to insert into the disequality store. When this expression is evaluated in any state with a substitution say `S`, the result is `S` extended with `((y . 3) (x .  cat))` (assuming `(== x 3)` and `(== cat y)` do not already hold in `S`). Thus, we insert the extension `((y . 3) (x .  cat))` into the disequality store.

The following code implements the above case-wise analysis, with the auxiliary procedure `prefix-S` used to compute the extension of a substitution:

```
(define (=/= u v)
  (lambda (st)
    (let ([S (S-of st)]
          [C (C-of st)]
          [D (D-of st)])
      (cond
        [(unify u v S) =>
         (post-unify-=/= S C D)]
        [else (unit st)]))))

(define (post-unify-=/= S C D)
  (lambda (S+)
    (cond
      [(eq? S+ S) mzero]
      [else (let ([d (prefix-S S+ S)])
              (unit (make-st S C
                      (cons d D))))])))
```

The implementation of `prefix-S` makes use of the fact that whenever `unify` extends a substitution, `unify` always adds new pairs as prefixes to the existing substitution via `cons`. Thus, to find the extension of a substitution `S+` with respect to `S`, we simply take the prefix of `S+` which when removed makes `S+` equal to `S`.

```
(define (prefix-S S+ S)
  (cond
    [(eq? S+ S) '()]
    [else (cons (car S+) (prefix-S (cdr S+)
                                    S))]))
```

2.1.3 Unification and Disequality constraints. Next, we have to deal with the interaction between `==` and `=/=` constraints. Here, there are two possible cases:

1. a `==` constraint may violate an existing `=/=` constraint; or,
2. a `==` constraint may simplify an existing `=/=` constraint.

For the first case, we have to ensure that unifications do not violate any disequality constraints. A disequality constraint is violated in a substitution if all of its atomic disequalities are violated in the substitution. We can check if an atomic disequality between two terms is violated in a substitution by attempting to unify the two atomic terms in the substitution — if unification succeeds without extending the substitution, then the atomic disequality is violated. Thus, to check if a disequality constraint `d` is violated in a substitution `S` we check if each of `d`'s atomic disequalities are violated in `S`.

For example, consider the following state:

```
(() 2 (((y . 'cat)) ((x . 5) (y . 3))))
```

Here, `x` and `y` are placeholders for logic variables, the substitution is the empty substitution, the value of two for the counter indicates two logic variables have been introduced so far, and the disequality store holds the disequality constraints introduced so far. Now, if the following == constraint were to be applied to this state:

```
(== `(,x cat) `(5 ,y))
```

the new substitution would be:

```
((y . 'cat) (x . 5))
```

In this new substitution, all the atomic disequalities of the disequality constraint `((y . 'cat))` are violated as the logic variable `y` unifies with the symbol `cat` without extending the substitution. Thus, the above unification violates the first disequality constraint in the store.

Similarly, if the following == constraint were to be applied:

```
(== `(,x 3) `(5 ,y))
```

the new substitution would be:

```
((y . 3) (x . 5))
```

In the new substitution, all the atomic disequalities of the disequality constraint `((x . 5) (y . 3))` are violated as the logic variable `x` unifies with `5` and the logic variable `y` unifies with `3` without extending the substitution. Thus, the above == constraint violates the second disequality constraint in the store.

On the other hand, a == constraint such as:

```
(== `(,x 5) `(3 ,y))
```

would not violate any existing disequality constraints, as the extended substitution:

```
((y . 5) (x . 3))
```

would not result in the violation of all the atomic disequalities of any disequality constraint.

On the other hand, a == constraint may instead simplify some of the disequality constraints in the store. As an example consider the following program fragment:

```
(fresh (x y)
  (=/= `(,x 3) `(cat ,y))
  (== x cat))
```

Without the == constraint, the disequality constraint would have been `((y . 3) (x . cat))`. But after the == constraint, the disequality constraint simplifies to `((y . 3))` as the logic variable `x` has already been unified with `cat`. The key idea behind simplification is that we may discard those atomic disequalities in a disequality constraint which no longer hold with the == constraint.

We go through each of the disequality constraints in the disequality constraint store and check for both violation and simplification in the new substitution.

```
(define (reform-D D D^ S)
  (cond
    [(null? D) D^]
    [(unify* (car D) S) =>
     (lambda (S^)
       (cond
         [(eq? S S^) #f]
         [else (let ([d (prefix-S S^ S)])
                 (reform-D (cdr D)
                           (cons d D^)
                           S))]))]
    [else (reform-D (cdr D) D^ S)]))
```

The auxiliary procedure `unify*` attempts to unify the atomic terms of each atomic disequality in a disequality constraint `d` with respect to a substitution `S`. If `unify*` succeeds, `unify*` returns an extended substitution where the atomic terms of each disequality are unified. Otherwise, `unify*` returns `#f` to indicate failure. The extension of the returned substitution is the simplified disequality constraint. If `d` is violated, the extension is the empty list and the returned substitution is same as `S`. Thus, we can use `unify*` for both checking violation and simplification.

```
(define (unify* d S)
  (cond
    [(null? d) S]
    [(unify (caar d) (cdar d) S) =>
     (lambda (S)
       (unify* (cdr d) S))]
    [else #f]))
```

We update the definition of == constraint to use `reform-D`.

```
(define (== u v)
  (lambda (st)
    (==-verify (unify u v (S-of st)) st)))

(define (==-verify S+ st)
  (cond
    [(not S+) mzero]
    [(eq? (S-of st) S+) (unit st)]
    [(reform-D (D-of st) '() S+) =>
     (lambda (D)
       (unit (make-st S+ (C-of st) D)))]
    [else mzero]))
```

2.1.4 Reification. Recall from microKanren that reification is the process by which the user sloughs off information from the resulting stream in order to clarify the presentation of that which is desired [Hemann and

Friedman 2013]. For reifying disequality constraints, we require the following properties:

1. every semantically equivalent program should produce the same answer, irrespective of the constraint order in the program; and,
2. irrelevant and redundant constraints must not be reified in the final answer.

Before dealing with these properties, we need to set up reification of disequality constraints. In particular, we need to reify logic variables involved in disequality constraints as the symbols they are mapped to in the substitution created by reify-S (e.g. the zeroeth logic variable as _.0). To do this, we walk* the disequality store in the substitution created by reify-S.

```
(define (reify-1st st*)
  (map (reify-var-state (var 0)) st*))

(define ((reify-var-state v) st)
  (let ([S (S-of st)]
        [D (D-of st)])
    (let ([v (walk* v S)]
          [D (walk* D S)])
      (let ([r (reify-S v '())])
        (let ([v (walk* v r)]
              [D (walk* (drop-dot-D D) r)])
          (prettify v D r))))))
```

drop-dot-D turns the pairs into lists so that the reified constraints do not have dots, making them easier to read.

```
(define (drop-dot-D D)
  (map (lambda (d)
         (map (lambda (d-pr)
                (let ([x (lhs d-pr)]
                      [u (rhs d-pr)])
                  `(,x ,u)))
              d))
       D))
```

prettify does the following to ensure the first reification property.

1. Lexicographically sort the atomic disequalities of each disequality constraint in D.
2. Lexicographically sort the disequality constraints in D.

```
(define (prettify v D r)
  (let ([D (sorter (map sorter D))])
    (cond
      [(null? D) v]
      [else `(,v (=/= . ,D))])))
```

The auxiliary procedure sorter is used to lexicographically sort a list. The Racket procedure display takes two arguments — a datum and an output port. It then displays the datum to the output port in such a way that byte- and character-based datatypes are written as raw bytes or characters (see Flatt and PLT [2022]).

```
(define (sorter ls) (sort ls lex<?))

(define (lex<? t1 t2)
  (let ([t1 (datum->string t1)]
        [t2 (datum->string t2)])
    (string<? t1 t2)))

(define (datum->string d)
  (let ([op (open-output-string)])
    (begin (display d op)
           (get-output-string op))))
```

For the second reification property, we have to remove irrelevant constraints. For example, consider the following program:

```
(run* (q) (== 'cat q) (fresh (x) (=/= 5 x)))
```

Here, the disequality constraint on the fresh logic variable x is irrelevant to the final answer of the query logic variable q. We call this optimization "purify", and the key idea behind implementing purify is this — if a disequality constraint d involves an atomic disequality between a fresh logic variable not in the final answer and any other term, then we may discard d. This is because a convenient value can always be assigned to the fresh logic variable to satisfy d, without affecting the final answer. In the above example, as x is a fresh logic variable not in the final answer, we may pick something other than 5 for x and satisfy the disequality constraint.

```
(define (purify-D D* r)
    (cond
      [(null? D*) '()]
      [(anyvar? (car D*) r)
       (purify-D (cdr D*) r)]
      [else (cons (car D*)
                  (purify-D (cdr D*) r))]))

(define (anyvar? v r)
  (cond
    [(var? v) (var? (walk v r))]
    [(pair? v) (or (anyvar? (car v) r)
                   (anyvar? (cdr v) r))]
    [else #f]))
```

To satisfy the second property, we must also avoid redundancy by discarding disequality constraints that are *subsumed* by other disequality constraints. We say that a constraint d1 subsumes d2 (or d2 is subsumed by d1) if whenever d1 holds, d2 also holds. For example consider the following program fragment:

```
(fresh (x y)
  (=/= 3 x)
  (=/= `(,x cat) `(3 ,y)))
```

If the first disequality constraint `(=/= 3 x)` holds, then the second constraint should also hold. Therefore, the first constraint subsumes the second constraint and the second constraint can be safely discarded. The important observation to make here is that `d1` subsumes `d2` if every atomic disequality in `d1` is also an atomic disequality in `d2` (with possibly more atomic disequalities not in `d1`).

In terms of implementation, `d1` subsumes `d2` if we can `unify*` `d1` by treating `d2` as a substitution (which we can, since disequality constraints are also association lists) without extending `d2`. This is because if each atomic disequality in `d1` is contained in `d2`, unifying the terms of atomic disequalities by treating `d2` as a substitution should not require extending `d2`. The following code implements this:

```
(define (rem-subsumed-D<D D D^)
  (cond
    [(null? D) D^]
    [(or (subsumed? (car D) D^)
         (subsumed? (car D) (cdr D)))
     (rem-subsumed-D<D (cdr D) D^)]
    [else
    (rem-subsumed-D<D (cdr D)
                      (cons (car D) D^))]))

(define (subsumed? d D)
  (and (not (null? D))
       (or (eq? (unify* (car D) d) d)
           (subsumed? d (cdr D)))))
```

We incorporate these to changes into our reifier.

```
(define ((reify-var-state v) st)
  (let ([S (S-of st)]
        [D (D-of st)])
    (let ([v (walk* v S)]
          [D (walk* D S)])
      (let ([r (reify-S v '())])
        (let ([v (walk* v r)]
              [D
              (walk* (drop-dot-D
                       (rem-subsumed-D<D
                         (purify-D D r)
                         '()))
                     r)])
          (prettify v D r))))))
```

2.2 Type constraints

In this section, we extend our implementation to also support the type constraints `numbero` and `symbolo`. Our programs are now of the form:

```
⟨goal-expr⟩ ::= (== ⟨term-expr⟩ ⟨term-expr⟩)
             |  (fresh (⟨id⟩+) ⟨goal-expr⟩+)
             |  (conde (⟨goal-expr⟩+)+)
             |  (=/= ⟨term-expr⟩ ⟨term-expr⟩)
             |  (numbero ⟨term-expr⟩)
             |  (symbolo ⟨term-expr⟩)
  Rest of the grammar remains the same.
```

Once again, using our recipe, implementing type constraints can be broken down into four steps.

1. Extend the state to include a type constraint store.
2. Define `numbero` and `symbolo` goal constructors that update the state.
3. Now we have two interactions to a deal with — between type and `==` constraints, and between type and disequality constraints. We need to:
   a. verify after each `==` constraint that no type constraints are violated, and possibly simplify the type constraint store.
   b. implement subsumption of disequality constraints by type constraints;
4. Update the reifier to display information from the type store.

2.2.1 Extending the State.

Firstly, we extend the state to hold a type constraint store.

```
(define (make-st S C D T)
  `(,S ,C ,D ,T))

...

(define (T-of st)
  (cadddr st))

(define empty-state (make-st '() 0 '() '()))
```

The type constraint store is a list of constraints, where each constraint consists of three components:

1. the logic variable on which the constraint exists (in our implementation type constraints on constant terms such as numbers and symbols get immediately resolved and are never added to the constraint store);
2. a tag naming the constraint, which will be useful while displaying the final answer;
3. a predicate corresponding to the constraint which can be applied to terms to see they satisfy the constraint.

Thus, for example if we were to apply the `numbero` constraint on a logic variable `x`, the generated constraint would be:

```
(x . (num . number?))
```

Here, `x` would be replaced by the actual logic variable it is represented by, the symbol `num` is the tag used to represent the `numbero` constraint, and the predicate `number?` can be applied to terms to see if they satisfy they constraint.

2.2.2 Defining the constraints. As both `symbolo` and `numbero` are quite similar, we implement both using the common goal constructor `make-type-constraint`. It constructs a goal given a tag, a predicate, and a term. The implementation of a type constraint can be broken down into three cases, each corresponding to a different type of invocation.

1. When applied to a logic variable, we first need to check if there are any disjoint type constraints already on that logic variable (e.g. applying both `symbolo` and `numbero` to the same logic variable must lead to no answers). If not, then this constraint must be added to the type constraint store (unless the same constraint already exists).
2. When applied to a pair, then no type constraint can hold as our type constraints only operate on atomic terms.
3. When applied to a constant term (e.g number or symbol) we may immediately apply the predicate corresponding to the constraint to determine whether the constant satisfies the constraint.

```
(define (make-type-constraint tag pred)
  (lambda (u)
    (lambda (st)
      (let ([S (S-of st)]
            [C (C-of st)]
            [D (D-of st)]
            [T (T-of st)]
            [A (A-of st)])
        (let ([u (walk u S)])
          (cond
            [(var? u)
             (cond
               [(make-type-constraint/x
                      u tag pred st S C D T A)
                      => unit]
               [else mzero])]
            [(pair? u) mzero]
            [else (cond
                    [(pred u) (unit st)]
                    [else mzero])]))))))

(define symbolo
  (make-type-constraint 'sym symbol?))

(define numbero
  (make-type-constraint 'num number?))
```

The auxiliary procedure `make-type-constraint/x` attempts to construct a type constraint on the logic variable `x` in a given state `st`. As discussed, when a type constraint is applied to a logic variable, we have to check for two cases — if the same constraint already exists on the logic variable (in which case we return the state as is), else if there is a disjoint type constraint on the logic variable (in which case we fail and return the empty stream as there are no answers). We implement this via the auxiliary procedure `ext-T`, which goes through the type constraints in the type store and for each type constraint, checks for both the cases.

```
(define (ext-T x tag pred S T)
  (cond
    [(null? T) `((,x . (,tag . ,pred)))]
    [else (let ([t (car T)]
                [T (cdr T)])
            (let ([t-tag (tag-of t)])
              (cond
                [(eq? (walk (lhs t) S) x)
                 (cond
                   [(tag=? t-tag tag) '()]
                   [else #f])]
                [else
                 (ext-T x tag pred S T)])))]))
```

We can now use `ext-T` to define `make-type-constraint/x`:

```
(define make-type-constraint/x
  (lambda (u tag pred st S C D T)
    (cond
      [(ext-T u tag pred S T) =>
       (lambda (T+)
         (cond
           [(null? T+) st]
           [else (let ([T (append T+ T)])
                   (make-st S C D T))]))]
      [else #f])))
```

2.2.3 Unification and Type Constraints. Next, similar to disequality constraints, we have to deal with the interaction between unification and type constraints. Not only do we have to check that unifications do not violate type constraints, but also possibly simplify type constraint store. For example, consider the following program fragment:

```
(fresh (x)
  (symbolo x)
  (== 5 x))
```

The unification here breaks the `symbolo` constraint on `x`. Furthermore, unifications may also simplify the constraint store. For instance:

```
(fresh (x)
  (numbero x)
  (== 10 x))
```

After unification, we may discard the `numbero` constraint.

To implement these two, we go through each type constraint and check for both the cases — whether the new unification broke the type constraint or whether the type constraint may be discarded. If neither, we return false to indicate the constraint is retained as is.

```
(define (reform-T T S)
  (cond
    [(null? T) '()]
    [(reform-T (cdr T) S) =>
     (lambda (T0)
       (let ([u (walk (lhs (car T)) S)]
             [tag (tag-of (car T))]
             [pred (pred-of (car T))])
         (cond
           [(var? u)
            (cond
              [(ext-T u tag pred S T0) =>
               (lambda (T+)
                 (append T+ T0))]
              [else #f])]
           [else (and (pred u) T0)])))]
    [else #f]))
```

We now update `==-verify` to use `reform-T`, and to remove any disequality constraints subsumed by the reformed type store.

```
(define (==-verify S+ st)
  (cond
    ...
    [(reform-D (D-of st) '() S+) =>
     (lambda (D)
       (cond
         [(reform-T (T-of st) S+) =>
          (lambda (T)
            (unit
              (make-st S+
                       (C-of st)
                       (rem-subsumed-D<T T D)
                       T)))]
         [else mzero]))]
    ...))
```

2.2.4 Implementing Disequality Subsumption.

We also have to deal with the interaction between disequality constraints and type constraints. In particular, a type constraint may let us discard certain disequality constraints. For example, consider the following program fragment:

```
(fresh (a) (=/= 'cat a) (numbero a))
```

The disequality constraint can be safely discarded as there is no number that is the symbol `cat`.

As a more complicated example, consider:

```
(fresh (x y)
  (=/= '(cat dog) `(,x ,y))
  (numbero x))
```

Here, the disequality constraint consists of two disequalities — between `cat` and `x`, and `dog` and `y`. As the logic variable `x` is constrained to be a number by the `numbero` constraint, `x` can never be `cat` and thus we can safely discard the disequality constraint.

However, we must be careful not to discard disequality constraints in cases such as the following:

```
(fresh (a) (=/= 'cat a) (symbolo a))
```

Here, although the logic variable `a` is constrained to be a symbol, `a` still cannot be the symbol `cat` owing to the disequality constraint. This is something not captured by the type constraint, hence we cannot discard the disequality constraint.

The key idea here is that a disequality constraint `d` is subsumed by a type constraint `t` if `t` constraints a logic variable `x` to a type disjoint from the type of term `x` is not allowed to be in `d`.

We implement subsumption by removing all disequality constraints having a disequality between a type constrained logic variable and a ground term not satisfying the type constraint. Such disequality constraints can safely be discarded as the type constraint on the logic variable ensures that unifying the logic variable with values not satisfying the type constraint leads to failure. We use the auxiliary procedure `subsumed-d-pr?` to check if a disequality `d-pr` from a disequality constraint is subsumed by any type constraint in a type store `T`.

```
(define (subsumed-d-pr? T)
  (lambda (d-pr)
    (let ([u (rhs d-pr)])
      (cond
        [(var? u) #f]
        [else
          (let ([sc (assq (lhs d-pr) T)])
            (and sc
                 (let ([tag (tag-of sc)])
                   (cond
                     [((pred-of sc) u) #f]
                     [else #t]))))]))))
```

We use the Racket procedure `findf` to implement subsumption. `findf` takes two arguments, a predicate and a list, and returns the first element in the list satisfying the predicate or `#f` if no such element exists. For each disequality constraint `d` in the disequality store `D`, we use `findf` to check if `d` has a disequality that is subsumed by any type constraint in the store `T`. If so, we remove `d` from `D`.

```
(define (rem-subsumed-D<T T D)
  (filter (lambda (d)
            (not (findf (subsumed-d-pr? T) d)))
          D))
```

We update `make-type-constraint+` to employ subsumption:

```
(define make-type-constraint/x
  ...
       [else (let ([T (append T+ T)]
                   [D (rem-subsumed-D<T T+
                                         D)])
               (make-st S C D T))]))]
  ...))
```

2.2.5 Reification. Again, as with `=/=` we filter out constraints not relevant to the final answer during reification. Any type constraint on a fresh logic variable not in the final answer may be discarded as the type constraint can always be satisfied by assigning the fresh logic variable a value satisfying the constraint.

```
(define ((reify-var-state v) st)
  (let ([S (S-of st)]
        [D (D-of st)])
    (let ([v (walk* v S)]
          [D (walk* D S)])
      (let ([r (reify-S v '())])
        (let ([v (walk* v r)]
              [D
               (walk*
                 (drop-dot-D
                   (rem-subsumed-D<D
                     (purify-D D r)
                     '()))
                   r)]
              [T
               (walk*
                 (drop-pred-T
                   (purify-T T r))
                 r)])
          (prettify v D T r))))))

(define (purify-T T r)
  (filter (lambda (t)
            (not (var? (walk (lhs t) r))))
          T))

(define (drop-pred-T T)
  (map (lambda (t)
         (let ([x (lhs t)]
               [tag (tag-of t)])
           `(,tag ,x)))
       T))
```

In addition to the previous properties for reification, we require one additional property: logic variables with the same constraint must be grouped together for improved readability. For example:

```
> (run* (x) (fresh (a b c)
              (== (list a b c) x)
              (symbolo a)
              (numbero b)
              (symbolo c)))
'(((_.0 _.1 _.2) (num _.1) (sym _.0 _.2)))
```

Here, the logic variables with the `symbolo` constraint are grouped together.

To implement grouping, we repeatedly group all elements having the same constraint tag as the first element in the constraint store until there are no more elements in the constraint store. Additionally, as with disequality constraints, we sort lexicographically each part as well as the entire partition by tag to ensure the same answer for all semantically equivalent programs.

```
(define (prettify v D T r)
  (let ([D (sorter (map sorter D))]
        [T (sorter (map sort-part
                        (partition* T)))])
    (cond
      [(and (null? D) (null? T)) v]
      [(null? D) `(,v . ,T)]
      [(null? T) `(,v (=/= . ,D))]
      [else `(,v (=/= . ,D) . ,T)])))

(define partition*
  (lambda (A)
    (cond
      ((null? A) '())
      (else
       (part (lhs (car A)) A '() '())))))

(define part
  (lambda (tag A x* y*)
    (cond
     ((null? A)
      (cons `(,tag . ,(map car x*))
            (partition* y*)))
     ((tag=? (lhs (car A)) tag)
      (let ((x (rhs (car A))))
        (let ((x* (cond
                    ((memq x x*) x*)
                    (else (cons x x*)))))
          (part tag (cdr A) x* y*))))
     (else
      (let ((y* (cons (car A) y*)))
        (part tag (cdr A) x* y*))))))

(define (sort-part pr)
  (let ((tag (car pr))
        (x* (sorter (cdr pr))))
    `(,tag . ,x*)))
```

2.3 absento

The final constraint we implement is `absento`. We implement a restricted version of `absento`, where we require the first argument to be a ground symbol (although the second argument can be an arbitrary term expression). Our programs are now of the form:

```
⟨goal-expr⟩ ::= (== ⟨term-expr⟩ ⟨term-expr⟩)
              | (fresh (⟨id⟩+) ⟨goal-expr⟩+)
              | (conde (⟨goal-expr⟩+)+)
              | (=/= ⟨term-expr⟩ ⟨term-expr⟩)
              | (numbero ⟨term-expr⟩)
              | (symbolo ⟨term-expr⟩)
              | (absento (quote ⟨symbol⟩)
                ⟨term-expr⟩)

⟨symbol⟩    ::= Any valid Racket symbol
    Rest of the grammar remains the same.
```

Implementing `absento` can be broken down into four steps, in accordance with our recipe.

1. Extend the state to include a constraint store for `absento` constraints. We refer to the constraints generated by `absento` as ***absence*** constraints, and the constraint store as ***absence constraint store***.
2. Define the `absento` goal constructor to update the absence constraint store.
3. Now we have three interactions to a deal with — between absence and unification constraints, between absence and disequality constraints, and between absence and type constraints. We need to:
   a. verify after each successful unification that no absence constraints are violated, and possibly simplify the absence constraint store;
   b. discard any disequality constraints subsumed by absence constraints;
   c. use type constraint information to reduce absence constraints to disequality constraints.
4. Update the reifier to display information from the disequality store.

2.3.1 Extending the State.

Firstly, we extend the state to contain an absence constraint store.

```
(define (make-st S C D T A)
  `(,S ,C ,D ,T ,A))

...

(define (A-of st)
  (caddddr st))
```

```
(define empty-state (make-st '() 0 '() '() '()))
```

The absence constraint store contains a list of constraints, where each constraint contains three components.

1. The logic variable on which the constraint exists. In our implementation absence constraints on constant terms (such as numbers, symbols, and pairs consisting of only constants) get immediately resolved and are never added to the constraint store.
2. A tag naming the constraint, which will be useful while displaying the final answer.
3. A predicate corresponding to the constraint which can be applied to terms to see they satisfy the constraint.

2.3.2 Defining the Constraint.

To implement `absento`, we first check if its invocation was valid.

```
(define (absento tag u)
  (cond
    [(not (tag? tag))
     (error
       "Incorrect absento usage: ~s is not a tag"
       tag)]
    [else
      (lambda (st)
        (let ([S (S-of st)]
              [C (C-of st)]
              [D (D-of st)]
              [T (T-of st)]
              [A (A-of st)])
          (cond
            [(absento/u u tag st S C D T A) =>
             unit]
            [else mzero])))]))
```

Then, we do a case-wise analysis on its second argument. If the second argument is:

1. a number or a symbol we just need to check for disequality;
2. a logic variable we extend the absence store unless the same constraint already exists in the store;
3. a pair we recur on the `car` and the `cdr` which may add more constraints to the store.

We end up with the following code for the case-wise analysis.

```
(define (absento/u u tag st S C D T A)
  (let [(u (walk u S))]
    (cond
      [(var? u)
       (let ([A+ (ext-A u tag S T)])
         (cond
           [(null? A+) st]
           [else
```

```
             (unit
               (make-st S
                        C
                        D
                        T
                        (append A+ A)))]))]
       [(pair? u)
        (let ([au (car u)]
              [du (cdr u)])
          (let ([st
                 (absento/u
                   au tag st S C D T A)])
            (and st
                 (let ([S (S-of st)]
                       [C (C-of st)]
                       [D (D-of st)]
                       [T (T-of st)]
                       [A (A-of st)])
                   (absento/u
                     du tag st S C D T A)))))]
       [else
        (cond
          [(and (tag? u) (tag=? u tag)) #f]
          [else st])])))
```

Before extending the absence store, we first check if the constraint being inserted already exists in the store. We go through each constraint in the absence store and check if any of them are the same as the constraint being inserted.

```
(define (ext-A x tag S A)
  (cond
    [(null? A)
     (let ([pred (make-pred-A tag)])
       `((,x . (,tag . ,pred))))]
    [else
      (let ([a (car A)]
            [A (cdr A)])
        (let ([a-tag (tag-of a)])
          (cond
            [(eq? (walk (lhs a) S) x)
             (cond
               [(tag=? a-tag tag) '()]
               (else ext-A x tag S A))]
            [(tag=? a-tag tag)
             (let ([a-pred (pred-of a)])
               (ext-A-with-pred
                 x tag a-pred S A))]
            [else (ext-A x tag S A)])))]))

(define (make-pred-A tag)
  (lambda (x)
    (not (and (tag? x) (tag=? x tag)))))
```

As an optimization we also make sure to never create a new absence predicate for a tag that already has a predicate in the constraint store. When we encounter such a predicate in the store, we invoke the auxiliary `ext-A-with-pred` by with the existing predicate as an argument to avoid creating a duplicate predicate for the same tag.

```
(define (ext-A-with-pred x tag pred S A)
  (cond
    [(null? A) `((,x . (,tag . ,pred)))]
    [else
      (let ([a (car A)])
        (let ([a-tag (tag-of a)])
          (cond
            [(eq? (walk (lhs a) S) x)
             (cond
               [(tag=? a-tag tag) '()]
               [else
                 (ext-A-with-pred
                   x tag pred S (cdr A))])]
            [else
              (ext-A-with-pred
                x tag pred S (cdr A))])))]))
```

2.3.3 Reduction of Constraints. An invariant we place upon our absence store is that there should be no absence constraints that can be reduced to a disequality constraint, or discarded entirely. This is motivated by the reification properties we require. For instance, consider the following program:

```
(run 1 (x)
  (absento 'cat x)
  (symbolo x))
> '((_.0 (=/= ((_.0 cat))) (sym _.0)))
```

Here, since the logic variable `x` is constrained to be a symbol, we can safely recast the absence constraint as a disequality between `x` and the symbol `cat`, while still constraining `x` to be a symbol. Instead, if we were to not reduce the absence constraint to a disequality, the answer to the query would have been:

```
(run 1 (x)
  (absento 'cat x)
  (symbolo x))
> '((_.0 (sym _.0) (absento (_.0 cat))))
```

which, while not incorrect, is not quite accurate as an absence constraint is much more general than a disequality constraint.

Similarly, in a case such as follows:

```
(fresh (x)
  (absento 'cat x)
  (numbero x))
```

We can altogether discard the absence constraint safely.

To implement reduction, we go through each logic variable with a type constraint and check if there are any absence constraints on the logic variable as well.

```
(define (absento->diseq A+ S C D T A)
  (let ([x* (remove-duplicates (map lhs T))])
    (absento->diseq+ x* A+ S C D T A)))

(define (absento->diseq+ x* A+ S C D T A)
  (cond
    [(null? x*)
     (let ([A (append A+ A)])
       (make-st S C D T A))]
    [else
      (let ([x (car x*)]
            [x* (cdr x*)])
        (let ([D/A
                (absento->diseq/x x S D T A+)])
          (let ([D (car D/A)]
                [A+ (cdr D/A)])
            (absento->diseq+
              x* A+ S C D T A))))]))
```

If so, we try the following in order:

1. discard the absence constraint;
2. reduce the absence constraint to a disequality constraint;
3. leave the absence constraint as is.

```
(define (absento->diseq/x x S D T A)
  (absento->diseq/x+ x '() S D A))

(define (absento->diseq/x+ x A+ S D A)
  (cond
    [(null? A)
     `(,D . ,A+)]
    [else
      (let ([a (car A)]
            [A (cdr A)])
        (cond
          [(eq? (lhs a) x)
           (let ([D (ext-D x (tag-of a) D S)])
             (absento->diseq/x+ x A+ S D A))]
          [else
            (let ([A+ (cons a A+)])
              (absento->diseq/x+
                x A+ S D A))]))]))
```

The auxiliary `ext-D` checks if a disequality constraint between the absence constraint's logic variable and tag already exists. If so, we may discard the absence constraint safely. Otherwise `ext-D` extends the disequality store by adding a disequality between the absence constraint's logic variable and tag.

```
(define (ext-D x tag D S)
  (cond
    [(findf (lambda (d)
              (and (null? (cdr d))
                   (let ([d-lhs (lhs (car d))]
                         [d-rhs (rhs (car d))])
                     (and
                       (eq? (walk d-lhs S) x)
                       (tag? d-rhs)
                       (tag=? d-rhs tag)))))
            D)
     ; Disequality already exists,
     D]
    ; Reduction to disequality constraint
    [else (cons `((,x . ,tag)) D)]))
```

2.3.4 Subsuming Disequality Constraints. Next, to keep the disequality store as simple as possible, we delete any disequality constraint that is subsumed by an absence constraint. For example,

```
(run 1 (x)
  (=/= x 'cat)
  (absento 'cat `(bat . ,x)))
```

Here, whenever the absence constraint holds, so does the disequality constraint. Therefore, we may discard the disequality constraint.

This can be implemented by a simple modification to the subsumption criteria in our implementation for subsumption of disequality constraints by type constraints. A disequality constraint is subsumed by an absence constraint if both of them involve the same logic variable, and the tag the logic variable is not allowed to be equal is same as the tag in the absence constraint.

```
(define (rem-subsumed-D<T/A T/A D)
  (filter (lambda (d)
            (not (findf (subsumed-d-pr? T/A)
                        d)))
          D))

(define (subsumed-d-pr? T/A)
  ...
        (let ([c (assq (lhs d-pr) T/A)])
          (and c
               (let ([tag (tag-of c)])
                 (cond
                   [(and (tag? tag)
                         (tag? u)
                         (tag=? u tag))]
                   ...))))]))))
```

With subsumption implemented, we also have to update our `absento` implementation to employ subsumption.

```
(define (absento/u u tag st S C D T A)
  (let [(u (walk u S))]
    (cond
    ...
           [else
             (let ([D
                     (rem-subsumed-D<T/A
                       A+ D)])
               (unit
                 (absento->diseq
                   A+ S C D T A)))]))]
    ...)))
```

2.3.5 Verifying Constraints' Validity. Similar to previous constraints, we next have to ensure that unifications do not break any absence constraints and perform simplifications, if any, resulting from unifications. We check for each absence constraint `a` if `a` can be discarded, or if `a` creates new absence constraints. For example, consider the following program fragment:

```
(fresh (x)
  (absento 'cat x)
  (== 'bat x))
```

In the above program fragment, when is `x` is unified with `bat`, the absence constraint is vacuously true, and may be discarded.

On the other hand, consider the following program fragment:

```
(fresh (x)
  (absento 'cat x)
  (fresh (y z)
    (== (cons y z) x)))
```

Here, if the symbol `cat` must be absent in the logic variable `x`, `cat` must also be absent in the logic variables `y` and `z`. Thus, the above absence constraint generates two more absence constraints.

In the implementation, we go through each absence constraint in the store:

```
(define (reform-A A S)
  (cond
    [(null? A) '()]
    [(reform-A (cdr A) S) =>
     (reform-A+ (lhs (car A)) A S)]
    [else #f]))
```

Then we walk the constrained logic variable `x` (as `x` may have been involved in unifications) and do a case-wise analysis for kind of term `x` walks to. If `x` walks to a fresh logic variable, we leave the absence constraint as is; if `x` walks to a pair, we recur on the `car` and `cdr` of the pair (to possibly generate more absence constraints); and if `x` walks to a constant, we apply the absence predicate to check if the term violates the constraint.

```
(define (reform-A+ x A S)
  (lambda (A0)
    (let ([u (walk x S)]
          [tag (tag-of (car A))]
          [pred (pred-of (car A))])
      (cond
        [(var? u)
         (cond
           [(ext-A-with-pred x tag pred S A0) =>
            (lambda (A+)
              (append A+ A0))])]
        [(pair? u)
         (let ([au (car u)]
               [du (cdr u)])
           (cond
             [((reform-A+ au A S) A0) =>
              (reform-A+ du A S)]
             [else #f]))]
        [else (and (pred u) A0)]))))
```

Finally, we update `==-verify` to call `reform-A`.

```
(define (==-verify S+ st)
  (cond
    ...
    [(reform-D (D-of st) '() S+) =>
     (lambda (D)
       (cond
         [(reform-T (T-of st) S+) =>
          (lambda (T)
            (cond
              [(reform-A (A-of st) S+) =>
               (lambda (A)
                 (unit (make-st
                         S+
                         (C-of st)
                         (rem-subsumed-D<T/A T D)
                         T
                         A)))]
    ...))
```

2.3.6 Reification. Once again, we remove absento constraints involving irrelevant free logic variables from the final answer. Since the structure of an absence constraint and a type constraint is the same, we reuse `purify-T` and `drop-pred-T` by renaming them to reflect their new purpose as `purify-T/A` and `drop-pred-T/A`.

```
(define ((reify-var-state v) st)
  (let ([S (S-of st)]
        [D (D-of st)])
    (let ([v (walk* v S)]
          [D (walk* D S)])
      (let ([r (reify-S v '())])
        (let ([v (walk* v r)]
              [D (walk* (drop-dot-D
                          (rem-subsumed-D
                            (purify-D D r)
                            T))
                        r)]
              [T (walk* (drop-pred-T/A
                          (purify-T/A T r))
                        r)]
              [A (walk* (drop-pred-T/A
                          (purify-T/A A r))
                        r)])
          (prettify v D T A r))))))

(define (prettify v D T A r)
  (let ([D (sorter (map sorter D))]
        [T (sorter (map sort-part
                        (group-types-T T)))]
        [A (sorter A)])
    (let ([AT (append (if (null? A)
                        '()
                        `((absento . ,A)))
                      T)])
      (cond
        [(and (null? D) (null? AT)) v]
        [(null? D) `(,v . ,AT)]
        [(null? AT) `(,v (=/= . ,D))]
        [else `(,v (=/= . ,D) . ,AT)]))))
```

3 Related work

microKanren [Hemann and Friedman 2013] demonstrates how to implement a minimal subset of miniKanren. We use this as the starting point for our implementation as it provides the implementation of the core functionality of a miniKanren implementation.

Hemann and Friedman [2017] present a framework based on macros for generating Kanren implementations supporting various constraints, given as input predicates for the constraints. While this discusses how to build up constraint stores corresponding to various constraints, it does not discuss how to solve these constraints or reify the constraint stores into a final answer.

Byrd [2009] discusses how to implement `=/=`, including reification and subsumption. Our implementation of the disequality constraint was based on this.

Finally, Byrd et al. [2012] briefly discusses how to implement the constraints we do in the appendix. As the focus of the paper is on relational interpreters, it does not serve as an ideal resource to learn about implementing the relational constraints themselves. Our implementation here however, was based on the code provided in the appendix of this paper.

Acknowledgments

We would like to thank Abhinav S Menon and the anonymous reviewers for feedback.

We acknowledge Hemann and Friedman [2013] and Appendix D from Byrd et al. [2012] for the code upon which our implementation is based.

Research reported in this publication was supported by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number OT2TR003435. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

References

Michael Ballantyne. 2021. faster-miniKanren. https://github.com/michaelballantyne/faster-miniKanren.

William E Byrd. 2009. *Relational programming in miniKanren: techniques, applications, and implementations.* Ph. D. Dissertation. Indiana University.

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

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 2012 Annual Workshop on Scheme and Functional Programming.* 8–29.

Matthew Flatt and PLT. 2022. The Racket Reference. https://docs.racket-lang.org/reference/Writing.html.

Jason Hemann and Daniel P Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming. http://webyrd. net/scheme-2013/papers/HemannMuKanren2013. pdf.*

Jason Hemann and Daniel P Friedman. 2017. A framework for extending microKanren with constraints. *arXiv preprint arXiv:1701.00633* (2017).