A Tutorial Reconstruction of miniKanren with Constraints

BHARATHI RAMANA JOSHI, IIIT Hyderabad, India

WILLIAM E. BYRD, University of Alabama at Birmingham, USA

After one learns to program in miniKanren, it is natural to want to understand how it is implemented. However, there is no resource to learn how to implement miniKanren with constraints such as `=/=`, `symbolo`, `numbero`, and `absento`. Furthermore, these are used to implement a wide class of interesting programs such as relational interpreters and relational type inferencers, making the need for such a resource all the more felt. This paper aims to be that resource.

Additional Key Words and Phrases: miniKanren implementation, Racket, relational programming

ACM Reference Format:
Bharathi Ramana Joshi and William E. Byrd. 2018. A Tutorial Reconstruction of miniKanren with Constraints. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 31 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 [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 [Friedman 2013] and implement our miniKanren on top of this.

2 IMPLEMENTATION

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

The high level syntax of runnable programs to begin with is:

```
program = (run* (var) goal goal ...)
        | (run n (var) goal goal ...)
goal = (== term-expr-1 term-expr-2)
     | (conde (goal goal ...) (goal goal ...) ...)
     | (fresh (var var ...) goal goal ...)
```

The formal syntaxes of all our languages are deferred to Appendix B for the sake of brevity.

One major change we make to microKanren 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.

This changed microKanren is provided in the appendix A.

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. To implement a new constraint `c`:

(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.
(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:

```
program = (run* (var) goal goal ...)
        | (run n (var) goal goal ...)
```

```
goal = (== term-expr-1 term-expr-2)
     | (conde (goal goal ...) (goal goal ...) ...)
     | (fresh (var var ...) goal goal ...)
     | (=/= term-expr-1 term-expr-2)
```

New additions are highlighted in boldface throughout this paper.

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 this store;
(3) verify after each new unification that no disequality constraints are violated, and possibly simplify the disequality constraint store;
(4) update the reifier to display information from the disequality store.

2.1.1 Extending the State. Firstly, we 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 `x` is unified with `cat` *and* `y` is unified with `3`. This is captured by the following association list:

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

That is, the pairs in an association list are those pairs whose `car`s and `cdr`s cannot be unified in conjunction. In the above example, the disequality constraint would be violated if `y` were unified with `3` in conjunction with `x` unified 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. To accomplish this we reuse `unify`. Consider `=/=` applied to terms say `t1` and `t2`. When we try to unify `t1` and `t2` in the current substitution, there are three cases possible:

(1) `unify` fails. In this case, there is no possible way of extending the current substitution to make `t1` and `t2` hold (if there was, `unify` would have returned this extended substitution). Thus, the disequality constraint can safely be discarded.
(2) `unify` succeeds without extending the current substitution. 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 substitution contains precisely all those pairs whose `car` and `cdr` must be equal for the unification to succeed. In other words, for the `=/=` constraint to hold, this extension must not hold! Thus, we add this extension to our disequality store.

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 without extending it.
(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 an extended substitution that has the list `((y . 3) (x .  cat))` prepended to it. Thus, we insert this extension into the disequality store.

The following code implements the above, with the auxiliary procedure `prefix-S` being 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, it 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 Verifying Constraints' Validity. 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;
(2) or, a `==` constraint may simplify an existing `=/=` constraint.

Firstly, we have to ensure that new `==` constraints do not violate any existing disequality constraints. To check this, we once again make use of `unify` — if the unification of the `car` and the `cdr` of all the pairs in a disequality constraint `d` in a substitution `S` succeeds without extending it, `d` does not hold 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 `==` goal were to be applied to this state:

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

the new substitution would be:

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

In this new substitution, the unifications of the `car` and the `cdr` of all the pairs of the second disequality constraint `((x . 5) (y . 3))` would succeed. Thus, the above `==` constraint violates one of the existing disequality constraints.

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

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

the new substitution would be:

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

In this new substitution, the unifications of the `car` and the `cdr` of all the pairs of the first disequality constraint `((y . cat))` would succeed. Thus, the above `==` constraint violates one of the existing disequality constraints.

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 successful unification of the `car` and the `cdr` of all the pairs in any disequality constraint.

We define the auxiliary procedure `unify*` to unify the `car` and the `cdr` of all the pairs in a disequality constraint in a given substitution. It returns an extended substitution where all the unifications succeed by going through each pair in the disequality constraint and attempting to unify the `car` and the `cdr` in the current substitution. If any of the unifications fails, it returns `#f` instead.

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

Other than violating an existing disequality constraint, new `==` constraints 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))
```

Before the `==` constraint, the disequality constraint would have been `((y . 3) (x . cat))`. But after the `==` constraint, the disequality simplifies to `((y . 3))` as `x` has already been unified with `3`. To perform this simplification we first perform `unify*` of a disequality constraint in the substitution resulting from the `==` constraint, say `S`. If we call the resulting substitution from `unify*` say `S+`, then the extension of `S+` with respect to `S` will be the reduced disequality constraint. This is because whatever unifications have taken place because of the `==` constraint will be in `S`, and thus will not be in the extension. As a result, the extension will contain only those pairs which are missing from `S`, but whose unification will result in the violation of the disequality constraint.

Thus, 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)]))
```

We now update the definition of our `==` 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. When implementing miniKanren, we transform programs into internal representations such as logic variables, streams, constraint stores, etc. However, when presenting the final answer to the program writer, presenting such internal representations would demand of the programmer to know the underlying implementation of the system. To avoid this, we take the internal representations and render them in a form that is easily comprehensible by the programmer without knowing the internal implementation details. This process of rendering the internal data structures of our implementation is called "reification".

When it comes to reifying the disequality store, we insist the following properties.

(1) A fresh variable involved in disequality constraints should be reified as the appropriate symbol, which is consistent with the symbol used in the answer set. For example, as the zeroth fresh variable is rendered as `_.0` in the answer set, it must also be rendered as `_.0` in the reified disequality store.
(2) Every semantically equivalent program should produce the same answer, irrespective of the constraint order in the program. We simply sort the constraints and constraint store lexicographically to ensure this.
(3) Irrelevant and redundant constraints must not be rendered in the final answer.

For the first property, we reuse the reification substitution built by `reify-S` in microKanren and abusively `walk*` the disequality store `D` in it as `walk*` recursively looks up the `car` and the `cdr` of a pair, and `D` is a list of pairs.

```
(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 and `prettify` formats the answers and information in the disequality store so that it may appear in a user-friendly manner. In particular, `prettify` does the following and helps with accomplishing the second goal.

(1) Translate the association lists in the disequality store into a list of lists.
(2) Sort each association list internally lexicographically.
(3) Sort all the association lists lexicographically.

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

(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))
```

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 **?**).

```
(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))))
```

Finally, we implement filtering out irrelevant and redundant constraints. Firstly, we can discard disequality constraints that do not affect the final answer. For example, consider the following program:

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

Here, the disequality constraint on the fresh variable `x` is irrelevant when it comes to the final answer of the query variable `q`. We call this optimization "purify", and the key idea behind implementing it is this — if a disequality constraint contains a pair where the `car` (resp., `cdr`) is a fresh variable, then we may discard this disequality constraint. This is because when the `car` (resp., `cdr`) is a fresh variable, we can always pick something for this fresh variable that is not equal to the `cdr` (resp., `car`) and satisfy the disequality constraint. Continuing the above example, as `x` is a fresh variable, 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]))
```

Secondly, we can discard 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 latter can be safely discarded. The important observation to make here is that `d1` subsumes `d2` if every pair in `d1` is also contained in `d2` (with possibly more pairs not in `d1`).

The key idea behind removing subsumed disequality constraints is `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 pair in `d1` is contained in `d2`, unifying them 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 two optimizations 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:

```
program = (run* (var) goal goal ...)
        | (run n (var) goal goal ...)
goal = (== term-expr-1 term-expr-2)
     | (conde (goal goal ...) (goal goal ...) ...)
     | (fresh (var var ...) goal goal ...)
     | (=/= term-expr-1 term-expr-2)
     | (numbero term-expr)
     | (symbolo term-expr)
```

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 them both using a common goal constructor we call `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 variable, we first need to check if there are any disjoint type constraints already on that variable (e.g. applying both `symbolo` and `numbero` to the same variable must lead to no answers). If not, then this constraint must be added to the store (unless the same constraint already exists in the store).
(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 variable, we have to check for two cases — if the same constraint already exists on the variable (in which case we return the state as is), else if there is a disjoint type constraint on the 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
    ; Ran out of type constraints without any conflicts, add new type constraint
    ; to the store.
    [(null? T) `((,x . (,tag . ,pred)))]
    [else (let ([t (car T)]
                [T (cdr T)])
            (let ([t-tag (tag-of t)])
              (cond
                ; Is the current constraint on x?
                [(eq? (walk (lhs t) S) x)
                 (cond
                   ; Is it same as the new constraint? Then do not extend the
                   ; store.
                   [(tag=? t-tag tag) '()]
                   ; Is it conflicting with the new constraint? Then fail.
```

```
                  [else #f])]
               ; The current constraint is not on x, continue going through
               ; rest of the constraints
               [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 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 Implementing Subsumption. There is an important optimization to be implemented here to avoid redundancy in the disequality constraint store — delete any disequality constraints that are subsumed by a type constraint. 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 variable `x` is constrained to be a number by the `numbero` constraint, 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))
```

We implement this by removing all disequality constraints containing a disequality between a type constrained logic variable and a value not satisfying the type constraint. Such disequality constraints can safely be discarded as the type constraint on the logic variable ensures that unifying the variable with values not satisfying the type constraint leads to failure. We use the auxiliary procedure `subsumed-d-pr?` to check if a pair from a disequality constraint `d-pr` 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
        ; We want the disequality to be between a variable and a constant, can
        ; ignore constraints between two variables.
        [(var? u) #f]
```

```
        [else
          (let ([sc (assq (lhs d-pr) T)])
            ; Check if the variable is type constrained
            (and sc
                 (let ([tag (tag-of sc)])
                   (cond
                     ; Check if the constant satisfies the type constraint
                     [((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 pair that is subsumed by any type constraint in the store `T`. If so, we remove this `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 use this optimization:

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

2.2.4 Verifying Constraints' Validity. Next, similar to disequality constraints, we have to ensure that no new unifications break any type constraints. 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 a type constraint or whether it may be discarded. If neither, we return false to indicate the constraint is retained as it 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 this 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]))]
    ...))
```

2.2.5 Reification. Again, as with `=/=` we filter out constraints not relevant to the final answer during reification. Any type constraint on a fresh variable may be discarded as it can always be satisfied by making the fresh 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 demands set out for reification, we have one additional demand : 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 variables with the `symbolo` constraint are grouped together.

To implement this, 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 symbol only (although the second argument can be an arbitrary term expression). Our programs are now of the form:

```
program = (run* (var) goal goal ...)
        | (run n (var) goal goal ...)
goal = (== term-expr-1 term-expr-2)
     | (conde (goal goal ...) (goal goal ...) ...)
     | (fresh (var var ...) goal goal ...)
     | (=/= term-expr-1 term-expr-2)
     | (numbero term-expr)
     | (symbolo term-expr)
     | (absento tag term-expr)
```

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

(1) Extend the state to include a `absento` constraint store;

(2) Define the `absento` goal constructor to update this store;
(3) Now we have three interactions to a deal with — between absento and `==` constraints, between absento and disequality constraints, and between absento and type constraints. We need to:
   (a) verify after each `==` constraint that no absento constraints are violated, and possibly simplify the absento constraint store;
   (b) discard any disequality constraints subsumed by `absento` constraints;
   (c) use type constraint information to reduce absento 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 `absento` 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 `absento` 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 `absento` constraints on constant variable 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 variable we extend the **absento** 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 absento store, we first check if the constraint being inserted already exists in the store. As an optimization we also make sure to never create a new **absento** predicate for a tag that already has a predicate.

```
(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 (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))])))]))

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

2.3.3 Reduction of Constraints. An important optimization to implement here for performance is the reduction of `absento` constraints to disequality constraints. For instance, consider the following program fragment:

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

Here, since `x` is constrained to be a symbol, we can safely recast the `absento` constraint as a disequality between `x` and the symbol `cat`. Similarly, in a case such as follows:

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

We can altogether discard the `absento` constraint safely.

To implement this, we go through each variable with a type constraint on it and check if there are any `absento` constraints on it as well. If so, we try to do one of the following, in that order:

(1) Discard the `absento` constraint.
(2) Reduce the `absento` constraint to a disequality constraint.
(3) Leave the `absento` constraint as it is.

```
(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))))]))

(define (absento->diseq/x x S D T A)
  (cond
    [(null? T)
     `(,D . ,A)]
    [else
      (let ([t (car T)])
        (cond
          [(and (eq? (lhs t) x)
                (or (tag=? (tag-of t) 'sym)
                    (tag=? (tag-of t) 'num)))
           (absento->diseq/x+ x '() S D A)]
          [else
            (absento->diseq/x x S (cdr T) 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))]))]))

(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)
     D]
    [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 `absento` constraint. For example,

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

Here, whenever the `absento` 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 subsumption implementation from the previous subsection. A disequality constraint is subsumed by an `absento` constraint if both of them involve the same variable, and the tag the variable is not allowed to be equal is same as the tag in the `absento` 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 this implemented, we also have to update our **absento** implementation to use the above optimization.

```
(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 new unifications do not break any existing **absento** constraints and perform simplifications, if any, resulting from new unifications.

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

(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)]))))

(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 free variables from the final answer. Since the structure of an `absento` constraint and a type constraint is the same, we reuse `drop-pred-T` by renaming it to reflect its new purpose as `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 [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

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.

]racketreference Matthew Flatt and PLT. [n. d.]. The Racket Reference. https://docs.racket-lang.org/reference/Writing.html.

Jason Hemann 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).

A MODIFIED MICROKANREN

```
#lang racket

(provide (all-defined-out))

(define (var c) (vector c))
(define var? vector?)
(define (var=? x1 x2) (= (vector-ref x1 0) (vector-ref x2 0)))

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

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

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

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

(define mzero '())
(define (unit st) (cons st mzero))

(define (ext-S u v S) `((,u . ,v) . ,S))

(define (walk u S)
  (let ([pr (and (var? u) (assoc u S var=?))])
    (if pr
      (walk (cdr pr) S)
      u)))

(define (unify u v S)
  (let ([u (walk u S)]
        [v (walk v S)])
    (cond
```

```
      [(and (var? u) (var? v) (var=? u v)) S]
      [(var? u) (ext-S u v S)]
      [(var? v) (ext-S v u S)]
      [(and (pair? u) (pair? v))
       (let ([S (unify (car u) (car v) S)])
         (and S (unify (cdr u) (cdr v) S)))]
      [else (and (eqv? u v) S)])))

(define (== u v)
  (lambda (st)
    (let ([S (unify u v (S-of st))])
      (if S (cons (make-st S (C-of st)) mzero) mzero))))

(define-syntax fresh
  (syntax-rules ()
    [(_ () g0 g ...) (conj+ g0 g ...)]
    [(_ (x0 x ...) g0 g ...)
     (call/fresh (lambda (x0) (fresh (x ...) g0 g ...)))]))

(define (call/fresh f)
  (lambda (st)
    (let ([C (C-of st)])
      ((f (var C)) (make-st (S-of st) (+ C 1))))))

(define (disj g1 g2)
  (lambda (st)
    (mplus (g1 st) (g2 st))))

(define (conj g1 g2)
  (lambda (st)
    (bind (g1 st) g2)))

(define (mplus $1 $2)
  (cond
    [(null? $1) $2]
    [(procedure? $1) (lambda () (mplus $2 ($1)))]
    [(pair? $1) (cons (car $1) (mplus (cdr $1) $2))]))

(define (bind $ g)
  (cond
    [(null? $) mzero]
```

```
    [(procedure? $) (lambda () (bind ($) g))]
    [(pair? $) (mplus (g (car $)) (bind (cdr $) g))]))

(define-syntax Zzz
  (syntax-rules ()
    [(_ g) (lambda (st) (lambda () (g st)))]))

(define-syntax disj+
  (syntax-rules ()
    [(_ g) (Zzz g)]
    [(_ g0 g ...) (disj (Zzz g0) (disj+ g ...))]))

(define-syntax conj+
  (syntax-rules ()
    [(_ g) (Zzz g)]
    [(_ g0 g ...) (conj (Zzz g0) (conj+ g ...))]))

(define-syntax conde
  (syntax-rules ()
    [(_ (g0 g ...) ...) (disj+ (conj+ g0 g ...) ...)]))

(define (pull $)
  (if (procedure? $) (pull ($)) $))

(define (take n $)
  (if (zero? n) empty
    (let ([$ (pull $)])
      (cond
        [(null? $) $]
        [else (cons (car $) (take (- n 1) (cdr $)))]))))

(define (take-all $)
  (let ([$ (pull $)])
    (if (null? $) $ (cons (car $) (take-all (cdr $))))))

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

(define ((reify-var-state v) st)
  (let ([v (walk* v (S-of st))])
    (walk* v (reify-S v '()))))
```

```
(define (reify-S v S)
  (let ([v (walk v S)])
    (cond
      [(var? v)
       (let ([n (reify-name (length S))])
         (cons `(,v . ,n) S))]
      [(pair? v) (reify-S (cdr v) (reify-S (car v) S))]
      [else S]))); number, bool

(define (reify-name n)
  (string->symbol
    (string-append "_." (number->string n))))

(define (walk* v S)
  (let ([v (walk v S)])
    (cond
      [(var? v) v]
      [(pair? v) (cons (walk* (car v) S)
                       (walk* (cdr v) S))]
      [else v])))

(define-syntax run*
  (syntax-rules ()
    [(_ (x) g0 g ...)
     (reify-1st (take-all (call/empty-state
                           (fresh (x) g0 g ...))))]))

(define-syntax run
  (syntax-rules ()
    [(_ n (x) g0 g ...)
     (reify-1st (take n (call/empty-state
                         (fresh (x) g0 g ...))))]))

(define (call/empty-state g) (g empty-state))
```

B FORMAL SYNTAXES

B.1 Modified microKanren

```
⟨program⟩ ::= (run <number> (⟨id⟩) ⟨goal-expr⟩)
            | (run* (⟨id⟩) ⟨goal-expr⟩)

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

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

⟨value⟩ ::= A Racket number or symbol

⟨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
```

The difference between term and values is that terms are unified with logic variables, but values may not be.

B.2 Adding disequality

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

Rest of the grammar remains the same.

B.3 Adding type constraints

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

$\quad | \quad$ `(conde` $(\langle goal\text{-}expr \rangle +)+$`)`
$\quad | \quad$ `(=/=` $\langle term\text{-}expr \rangle$ $\langle term\text{-}expr \rangle$`)`
$\quad | \quad$ `(numbero` $\langle term\text{-}expr \rangle$`)`
$\quad | \quad$ `(symbolo` $\langle term\text{-}expr \rangle$`)`

Rest of the grammar remains the same.

B.4 Adding absento

$\langle goal\text{-}expr \rangle$ ::= `(==` $\langle term\text{-}expr \rangle$ $\langle term\text{-}expr \rangle$`)`
$\quad | \quad$ `(fresh` $(\langle id \rangle +)$ $\langle goal\text{-}expr \rangle +$`)`
$\quad | \quad$ `(conde` $(\langle goal\text{-}expr \rangle +)+$`)`
$\quad | \quad$ `(=/=` $\langle term\text{-}expr \rangle$ $\langle term\text{-}expr \rangle$`)`
$\quad | \quad$ `(numbero` $\langle term\text{-}expr \rangle$`)`
$\quad | \quad$ `(symbolo` $\langle term\text{-}expr \rangle$`)`
$\quad | \quad$ `(absento` $\langle symbol \rangle$ $\langle term\text{-}expr \rangle$`)`

Rest of the grammar remains the same.