A Framework for Extending microKanren with Constraints *

Jason Hemann Daniel P. Friedman
Indiana University
{jhemann,dfried}@indiana.edu

Abstract

miniKanren is a family of embedded, domain-specific, relational (logic) programming languages. The microKanren approach implementing a miniKanren language–a small functional "kernel language" extended with straightforward macros–has been well received. The original microKanren spawned more than 50 implementations in more than 25 languages. Adding constraints beyond equality to a miniKanren language, however, has remained a complicated task. In recent implementations, adding a handful of additional constraints vastly increases the code's size and complexity. We describe instead the microKanren approach to adding constraints and present the Constraint microKanren framework for building constraint systems.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages, Constraint and logic languages

Keywords miniKanren, microKanren, constraint logic programming, relational programming, Racket

1. Introduction

Logic programming has proven to be a highly declarative approach to problem solving widely applicable to a broad class of problems [22]. The miniKanren family of languages is a group of embedded, domain-specific relational (logic) programming languages first presented in *The Reasoned Schemer* [10]. The miniKanren language has grown in popularity [33] and has found use both in academia and industry [3–5, 11, 28, 30].

In order to clarify the meaning and behavior of miniKanren programs, Hemann and Friedman introduced microKanren, a "kernel language" over which one can layer simple macros to develop a full miniKanren implementation [12]. One can program directly in microKanren, and at only 55 lines its implementation can serve as an object of study. Our original implementation has spawned at least 50 others in more than 25 languages, all of which can be found on miniKanren.org. Here, our aim is a similar kernel language augmented with a framework for constraints. This new implementation can be found at github.com/jasonhemann/constraint-microKanren.

The original miniKanren implementation contained only one constraint—equality (== in miniKanren). In recent implementations adding a handful of additional constraints vastly increased the code's size and complexity. The majority of that increase is due to the constraint management itself, including checking for violations, as well as constraint minimization and the presentation of constraints in answers (the latter have been termed "reification" in our implementations). Moreover, the constraint management is commingled with the flow of control, making such implementations more difficult to follow and diminishing their value as objects of study. The canonical Racket miniKanren implementation with these constraints is upwards of a thousand lines [27].

Here, we present the microKanren approach to adding constraints. We separate the control flow and variable introduction from the constraint management. We construct a framework in which to explore and add constraints, in which equality is but one (special) constraint. The control flow and variable introduction amounts to only 22 lines, and an entire implementation with the common miniKanren constraints beyond equality amounts to just over 100 lines. Briefly summarized, our contributions are:

- A new microKanren framework for constraints
- The clearest-yet-developed implementation of the typical miniKanren constraints
- An implementation of a constraint logic programming language still small enough to fit in your pocket.

In Section 2, we reintroduce miniKanren (microKanren) programming and the utility of constraints therein. In Section 3 we informally specify the responsibilities of the constraint architect. In Sections 4 and 5 we describe the constraints framework and implement typical miniKanren constraints. We also provide evidence of the robustness of our approach by implementing two new constraints `booleano` and `listo`. Finally, in Sections 6 and 7 we discuss related work and conclude.

2. miniKanren programming

We next briefly introduce miniKanren programming. The interested reader can find a more thorough explanation of miniKanren, defined in terms of microKanren, in Hemann and Friedman [12] and an introduction of an altogether different flavor in *The Reasoned Schemer* [10].

When programming in miniKanren, we describe the form of the solution; the implementation searches for terms or collections of terms satisfying the program's requirements. These requirements come in the form of *constraints*. This stands in contrast to programming directly in Scheme. Take for instance the `member` function. It expects a term `x` and a list `ls` and returns `ls`'s first sublist whose `car` is equal to `x`.

```
> (member 'x '(a x c))
'(x c)
```

The definition of the `membero` relation below is intended to approximate the behavior of the `member` function. The syntax we use here is a variation on that presented in *The Reasoned Schemer*.

```
(define-relation (membero x ls o)
  (fresh (a d)
    (== `(,a . ,d) ls)
    (conde
      ((== x a) (== ls o))
      ((membero x d o)))))
```

In the body of the goal constructor, we create two fresh variables, `a` and `d`, and ensure that `ls` is a pair. We also mandate that either of two situations hold: the first, in which both `x` is `a` and `ls` is `o`, or the second, in which `(membero x d o)` holds.

Programs in miniKanren are essentially the conjunction and disjunction of constraints which express requirements on, and relationships between, terms. At times auxiliary variables are needed to express these requirements. For instance, `(== `(,a . ,d) ls)` expresses a requirement that `ls` be some pair. The `run` and `run*` forms execute the program and process the answers. Our programs can have multiple answers, and so results are always presented in a list.

This implementation of `membero` acts as expected for the below translation of the previous invocation. The `run*` form returns all answers to a query.

```
> (run* (q) (membero 'x '(a x c) q))
'((x c))
```

Though our definition of `membero` only makes use of `==`, much recent research carried out in miniKanren relies on constraints beyond equality. Examples include a relational interpreter that doubles as a quine generator, a theorem prover that doubles as a proof assistant, and a type checker that doubles as a type inhabiter [4, 5, 28]. miniKanren with only the language forms used above is Turing complete, so in principle we could have computed the same functions without additional constraints. By saying research "relies" on these additional constraints, we mean that they allow the programmer to write interpreters, type checkers, etc. via a transformation from their functional implementations. In general, constraints enable more straightforward reasoning about the particular domain of the problem and thus facilitate clear code [16].

Quine generators and type inhabiters beautifully illustrate the benefits of additional constraints, but this is also clear with even a simple example. In the following call to `membero`, we receive an unexpected result. Where the `member` function returns the *first* sublist whose `car` matches the element, here we return *all* such sublists.

```
> (run* (q) (membero 'x '(a x x) q))
'((x x) (x))
```

To mirror the function's behavior using only the equality constraint, we must either resort to using some of the impure operators of miniKanren (such as `conda` or `condu`), adding some mechanism to perform `assert` and `retract` (à la Prolog) or else settle for the behavior of a `run 1` (`run` takes a parameter for the maximum number of answers to return).

To instead retain the function's behavior while remaining purely relational and maintaining the ability to `run` for multiple answers, we extend the language of miniKanren with disequality constraints. In miniKanren disequality constraints are introduced with the `=/=` operator and they behave similar to the `dif/2` constraint of various Prologs [35]. We modify the definition of `membero` and get the behavior we desire for the last query.

```
(define-relation (membero x ls o)
  (fresh (a d)
    (== `(,a . ,d) ls)
    (conde
      ((== x a) (== ls o))
      ((=/= x a) (membero x d o)))))

> (run* (q) (membero 'x '(a x x) q))
'((x x))
```

We get more interesting behavior as well. The next `run*` returns a list of two answers. Either the variable `y` is the symbol `x`, in which case `z` is `(x x)`, or it is some miniKanren term distinct from `x`, in which case `z` is `(x)`. In this second answer, `_.0` represents an arbitrary miniKanren term. When printed with the constraint `(=/= ((_.0 x)))`, the second answer means the variable `y` represents an arbitrary term other than `x`.

```
> (run* (q)
    (fresh (y z)
      (== q `(,y ,z))
      (membero 'x `(a ,y x) z)))
'((x (x x))
  ((_.0 (x)) (=/= ((_.0 x)))))
```

In constraint logic programming, answers are a collection of constraints. The analog to the most general unifier from logic programming is the most general collection of constraints. In miniKanren, the answers have been minimized and presented with respect to the query variable.

Constraints increase expressiveness in several important ways. Constraints provide a clear way for users to express the form of answers, as in the example above. Further, together with other features of logic programming languages, constraints can also help dramatically reduce the search space for possible solutions [26].

In addition, constraints compress what would be multiple answers (potentially infinitely many) into a single finite representation. Consider for instance, the `absento` constraint. An `absento` constraint holds between two terms `x` and `y` when `x` is neither equal to, nor a subterm of `y`. With just `=/=` constraints this relationship would be expressible, in general, only in the limit. This relationship is expressible only as an infinite conjunction of `=/=` constraints between `y` and terms from which `x` is absent. Adding `absento` as a constraint allows this relationship to be expressed finitely.

Researchers using miniKanren have developed constraints as needed. In addition to those mentioned above, other common constraints include the domain constraints `symbolo`, `numbero`, and `not-pairo`. These declare the constrained term to be a symbol, a number, or a non-pair respectively.

3. Constraint framework restrictions

Constraint microKanren is a framework for designing constraints and consequently constraint systems. It is similar to other CLP "shells" [25]. The constraint architect must specify if constraints are violated. The architect provides constraint-violation predicates to test for invalid sets of constraints. The architect must also provide the constraint names. From these, the framework will generate the microKanren (miniKanren) constraint operators and a constraint system automatically.

We fix the constraint domain to that of the miniKanren term language: symbols, logic variables, booleans, `()`, and

`cons` pairs of the above. In the interest of simplicity we reserve numbers as logic variables. We could have instead implemented variables using other data structures.

We demand that `==`, representing syntactic first-order equality, be a constraint of the system. It should be implemented by unification with the `occurs?` check. All constraints must be applicable over the entire term language, and must operate in all modes.

We also place restrictions on the constraint-violation predicates. Each should handle one category of violations. The predicates should be defined independently of the order in which they are invoked. That is, the predicates should be defined so that they may be tested in any order. Constraint-violation predicates, by definition, must be total functions. As such, the solver for the constraint system (`invalid?`, defined in Section 4) is also total.

The resultant constraint solver must be *well-behaved* [19]. This means it must be *logical*—that is, it gives the same answer for any representation of the same constraint information (i.e., regardless of order, redundancy, etc). It also must be *monotonic*—that is, for any set of constraints, if the solver reports that it is invalid, adding constraints cannot produce a valid set. Therefore, when adding a new constraint-violation predicate, a constraint architect is not required to redesign older ones. Doing so may, however, lead to clearer descriptions of the violations. Presently, all of the preceding restrictions are unchecked.

For us, the definition of a constraint system is the set of constraint interactions that are considered invalid. To specify these interactions via predicates in some sense *is* to define the constraints themselves.

Constraint violation is separate from constraint minimization. Here, we concern ourselves only with the former. Constraint microKanren performs no constraint minimization or answer projection [18]. In future work, we will use a similar approach for the minimization of the constraint store, answer projection, and the presentation of answers.

4. Constraints framework implementation

In this section we describe the implementation of Constraint microKanren's constraint framework. We model the constraint store with a persistent hash table. To install a new constraint in the store is to create a field in the hash table. Constraint microKanren uses the constraint's name as the key for its field in the store. The initial-state is constructed with something akin to `make-call/initial-state` below:

```
(define-syntax-rule (make-call/initial-state cid ...)
  (define S0 (make-immutable-hasheqv '((==) (cid) ...))))
```

Given a sequence of constraint identifiers, the posited new syntactic form `make-call/initial-state` would build a store with `==` and the other constraints, each associated with an empty list. Constraint identifiers must be unique, so each field has a distinct key. The different fields can also be seen as individual, distinct constraint stores; this is a common approach [34]. We use something like `make-call/initial-state` to construct the initial state when we make a constraint system.

Constraint goal constructors are created by invoking `make-constraint-goal-constructor` with the key of the constraint's corresponding field in the store. The function `make-constraint-goal-constructor` takes a field in the store, and returns a goal constructor.

```
(define (((make-constraint-goal-constructor key) . terms) S/c)
  (let ((S (ext-S (car S/c) key terms)))
    (if (invalid? S) '() (list `(,S . ,(cdr S/c))))))
```

The functions `ext-S` and `invalid?` are explained below. To construct the constraint itself, we globally define the constraint name as the result of invoking `make-constraint-goal-constructor`.

```
> (define == (make-constraint-goal-constructor '==))
> (define =/= (make-constraint-goal-constructor '=/=))
> (define symbolo (make-constraint-goal-constructor 'symbolo))
> (define absento (make-constraint-goal-constructor 'absento))
...
```

To constrain a term(s) during the execution of a program is to add the constrained term(s) to the corresponding field of the store. The `ext-S` function takes the store, the key, and the list of terms. The function adds those terms, as a data structure, to a list of such structures. The data structure is created by consing all of the terms together.

```
(define (ext-S S key terms)
  (hash-update S key ((curry cons) (apply list* terms))))
```

If the constraint store is consistent, we return a stream of a single state; if not, we return the empty stream. Once added, constraints are not removed from the store. This decision means the size of the constraint store and the cost of checking constraints grows with the number of times a particular constraint is encountered in the execution of a program. There are many ways to improve this approach.

Checking constraints takes place in `invalid?`. We use `make-invalid?` to build the definition of `invalid?`. The architect must provide a list containing a sequence of the constraint identifiers (except `==`). The architect must also provide a sequence of predicates that check for constraint violations. Each predicate should accept a substitution as an argument and return true if a violation is detected. The constraint identifiers should be free variables in the predicates. These variables will be bound in the expansion of `make-invalid?`. The result of `make-invalid?` is a predicate that tests if a store is invalid.

```
(define-syntax-rule (make-invalid? (cid ...) p ...)
  (λ (S)
    (let ((cid (hash-ref S 'cid)) ...)
      (cond
        ((valid-== (hash-ref S '==))
         => (λ (s) (or (p s) ...)))
        (else #t)))))
```

The first constraint we check is `==`. If this constraint is consistent, the result is a substitution. The substitution will be passed to each provided predicate when it is checked.

We used the phrase "something akin to" when describing `make-call/initial-state`. This is the main syntactic form for building constraint systems. The entire constraint system is built from one invocation of `make-constraint-system`. This new syntactic form takes the same parameters as does `make-invalid?`. It builds `invalid?`, the initial-state, and all of the constraints themselves. The result is a constraint system; together with microKanren's control infrastructure, this yields a full microKanren implementation (see Appendix).

The definition below uses Racket's `syntax-parse` [7]. We use `syntax-local-introduce` to introduce three new identifiers into lexical scope; the remaining constraint identifiers are already scoped.

```
(define-syntax (make-constraint-system stx)
  (syntax-parse stx
    [(_ (cid:id ...) p ...)
     (with-syntax
       ([invalid? (syntax-local-introduce #'invalid?)]
        [S0 (syntax-local-introduce #'S0)]
        [== (syntax-local-introduce #'==)])
       #'(begin
           (define invalid? (make-invalid? (cid ...) p ...))
           (define S0
             (make-immutable-hasheqv '((==) (cid) ...)))
           (define == (make-constraint-goal-constructor '==))
           (define cid (make-constraint-goal-constructor 'cid))
           ...))]))
```

5. Implementing a constraint system

Next, we implement a series of constraint-violation predicates, and their associated help functions, that together comprise a microKanren constraint system for a typical set of miniKanren constraints. We discuss these constraints and their predicates, one at a time.

The constraint == is included in every constraint system and is provided as part of the framework. The `valid-==` function below, and its associated help functions, is also included with the framework. The function expects a list of cons pairs of terms to be unified with each other. The definition of `unify` is provided in the Appendix.

```
(define (valid-== ==)
  (foldr
    (λ (pr s)
      (and s (unify (car pr) (cdr pr) s)))
    '()
    ==))
```

The == constraint is special because when deciding if constraints are violated, we treat terms of the language as classes quotiented by their meaning under the substitution. Assuming this field is valid, the resulting substitution is passed as an argument to the constraint-violation predicates. To construct a microKanren with just this constraint, the constraint architect should invoke `make-constraint-system` with an empty list of constraint identifiers and no constraint-violation predicates.

```
> (make-constraint-system
    ())
```

Beyond ==, our system contains four other constraints: `=/=`, `absento`, `symbolo`, and `not-pairo`. We discuss the predicates required to implement these constraints one at a time.

The first predicate tests for a violated `=/=` constraint. It searches for an instance where, with respect to the current substitution, two terms under a `=/=` constraint already unify. In that case, the `=/=` constraint has been violated.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo)
    (λ (s)
      (ormap
        (λ (pr) (same-s? (car pr) (cdr pr) s))
        =/=))
    ...)
```

It is implemented in terms of a help predicate `same-s?`. If the result of unifying two terms in the substitution is the same as the original substitution, then those terms were already equal relative to that substitution. We could have instead implemented `same-s?` by first checking to see if the call to `unify` has succeeded, and if so comparing the lengths of the substitutions.

```
#| Term × Term × Subst → Bool |#
(define (same-s? u v s) (equal? (unify u v s) s))
```

The next predicate checks for violated `absento` constraints, using the auxiliary predicate `mem?`. The predicate searches for an instance where, with respect to the substitution, the first term of a pair already unifies with (a subterm of) the second term. In that case, the `absento` constraint has been violated.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo)
    ...
    (λ (s)
      (ormap
        (λ (pr) (mem? (car pr) (cdr pr) s))
        absento))
    ...)
```

The predicate `mem?` checks if a term `u` is already equivalent to any subterm of a term `v` under a substitution `s`. It makes use of `same-s?` in the check. If the result of unifying `u` and `v` is the same as the substitution `s` itself, then the two terms are equivalent.

```
#| Term × Term × Subst → Bool |#
(define (mem? u v s)
  (let ((v (walk v s)))
    (or (same-s? u v s)
        (and (pair? v)
             (or (mem? u (car v) s)
                 (mem? u (cdr v) s))))))
```

We write a third constraint-violation predicate to search for a violated `symbolo` constraint. For each term under a `symbolo` constraint, we look if that term, relative to the substitution, is anything but a symbol or a variable. If so, that term violates the constraint. The function `walk` is defined in the Appendix.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo)
    ...
    (λ (s)
      (ormap
        (λ (y)
          (let ((t (walk y s)))
            (not (or (symbol? t) (var? t)))))
        symbolo))
    ...)
```

The predicate that checks for `not-pairo` violations works in a similar fashion. If any term with a `not-pairo` constraint, relative to the substitution, is a non-pair, non-variable term, then that term violates the constraint.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo)
    ...
    (λ (s)
      (ormap
        (λ (n)
          (let ((t (walk n s)))
            (not (or (not (pair? t)) (var? t)))))
        not-pairo)))
```

This completes the definition of the standard miniKanren constraints. When layering a miniKanren implementation over microKanren, the constraint operators are carried over

unchanged. To implement a complete microKanren with constraints, we would still need functions to minimize and present answers. This remains as future work.

Below is the execution of an example microKanren program that uses all of the constraints we have created. The result of invoking this program is a stream containing a single state. We can see that all the constraints are present in the constraint store, and we can read off each constraint. The `#hasheqv(...)` is the printed representation of the hash table, whose elements are the key/value pairs. For instance, the `=/=` field, `(=/= . ((c . 0) (0 . b)))`, contains the pairs `(c . 0)` and `(0 . b)`. These are the `=/=` constraints that have been added.

```
> (call/initial-state 1
    (call/fresh
      (lambda (x)
        (conj
          (== 'a x)
          (conj
            (=/= x 'b)
            (conj
              (absento 'b `(,x))
              (conj
                (not-pairo x)
                (conj
                  (symbolo x)
                  (=/= 'c x)))))))))
'((#hasheqv((== . ((a . 0)))
            (=/= . ((c . 0) (0 . b)))
            (absento . ((b 0)))
            (symbolo . (0))
            (not-pairo . (0)))
   .
   1))
```

5.1 Adding new constraints

To demonstrate the generality of this approach, we implement two constraints new for miniKanren: `booleano` and `listo`. The first mandates that the constrained term be a boolean, and the second a proper list. These constraints have more significant interactions than do the previous ones. As a result, we need several new predicates to support the implementation of these constraints.

These new constraints are interesting both because of their additional complexity, and also their utility. They can be used to improve the implementations of relational interpreters [5], an archetypal example of miniKanren programming. Consider the partially-completed miniKanren definition of the relational interpreter `val-ofo` below. This relation is intended to hold between an expression, an environment, and a value when that expression evaluates to the value in the environment.

```
(define-relation (val-ofo e env o)
  (conde
    ((symbolo e) (lookupo e env o))
    ((booleano e) (== e o) (listo env))
    ...))
```

If `e` is a variable, `o` is its value in the environment. We implement `lookupo` as a recursively-defined three-place relation. When the variable is found in the environment, we return its value. In prior implementations of relational interpreters, the remainder of the environment is left unconstrained. Without the `listo` constraint, the only way to ensure our environments are proper lists requires a recursive relation. This amounts to enumerating proper lists of all given lengths.

```
(define-relation (lookupo x ls o)
  (fresh (aa da d)
    (== ls `((,aa . ,da) . ,d))
    (conde
      ((== aa x) (== da o) (listo d))
      ((=/= aa x) (lookupo x d o)))))
```

Instead, we can now express infinitely many answers with a single `listo` constraint. We have also more tightly constrained the implementation of `lookupo`, which results in more precise answers.

In prior definitions of `val-ofo`, rather than using a `booleano` constraint, we equated the term first with `#t`, and then separately with `#f`. This generates near-duplicate programs that differ in their placement of `#t` and `#f`. By instead "compressing" the booleans into one, we ensure the programs we generate have a more interesting variety.

5.1.1 Implementing `booleano`

There are precisely two booleans, and this makes `booleano` more involved than other domain constraints. The first predicate checks that we haven't forbid a term from being `#t` and `#f` while demanding that it be a boolean. We also need a predicate to check for a `booleano`-constrained term that is a non-variable, non-boolean. Finally since the `booleano` domain constraint is incompatible with `symbolo`, the last predicate checks for terms constrained by both.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo booleano)
    ...
    (let ((not-b
            (λ (s)
              (or (ormap
                    (λ (pr) (same-s? (car pr) (cdr pr) s))
                    =/=)
                  (ormap
                    (λ (pr) (mem? (car pr) (cdr pr) s))
                    absento)))))
      (λ (s)
        (ormap
          (λ (b)
            (let ((s1 (unify b #t s)) (s2 (unify b #t s)))
              (and s1 s2 (not-b s1) (not-b s2))))
          booleano)))
    (λ (s)
      (ormap
        (λ (b)
          (let ((b (walk b s)))
            (not (or (var? b) (boolean? b)))))
        booleano))
    (λ (s)
      (ormap
       (λ (b)
         (ormap
           (λ (y) (same-s? y b s))
           symbolo))
       booleano)))
```

Below is an example of its use.

```
> (call/initial-state 1
    (call/fresh
      (lambda (x)
        (conj (=/= #f x) (conj (=/= #t x) (booleano x))))))
'()
```

5.1.2 Implementing `listo`

The `listo` constraint is more involved even than `booleano`, and consequently some of the constraint-violation predicates

are also more complex. We add four independent predicates to properly implement `listo`.

In the first of these, we look for an instance in which the end of something labeled a proper list is required to be a symbol. The function `walk-to-end` recursively walks the `cdr` of a term `x` in a substitution `s` and returns the final `cdr` of `x` relative to `s`. We use it in constraint-violation predicates related to the `listo` constraint.

```
#| Term × Subst → Bool |#
(define (walk-to-end x s)
  (let ((x (walk x s)))
    (if (pair? x) (walk-to-end (cdr x) s) x)))
```

The second predicate is similar to the first, except it checks for a boolean instead.

```
> (make-constraint-system
    (=/= absento symbolo not-pairo booleano listo)
    ...
    (λ (s)
      (ormap
        (λ (l)
          (let ((end (walk-to-end l s)))
            (ormap
              (λ (y) (same-s? y end s))
              symbolo)))
        listo))
    (λ (s)
      (ormap
        (λ (l)
          (let ((end (walk-to-end l s)))
            (ormap
              (λ (b) (same-s? b end s))
              booleano)))
        listo))
    (λ (s)
      (ormap
        (λ (l)
          (let ((end (walk-to-end l s)))
            (let ((s^ (unify end '() s)))
              (and s^
                   (ormap
                     (λ (n) (same-s? end n s))
                     not-pairo)
                   (or
                    (ormap
                      (λ (pr) (same-s? (car pr) (cdr pr) s^))
                      =/=)
                    (ormap
                      (λ (pr) (mem? (car pr) (cdr pr) s^))
                      absento))))))
        listo))
    (λ (s)
      (ormap
        (λ (l)
          (let ((end (walk-to-end l s)))
            (ormap
              (λ (pr)
                (and
                  (null? (walk (car pr) s))
                  (mem? end (cdr pr) s)))
              absento)))
        listo))
    ...)
```

In the third, we check for a proper list that must have a definite fixed last cdr (the `end`) under the substitution. This means either `end` already is `()`, or a `not-pairo` constrains `end`. If, in addition, either `=/=` or `absento` constraints forbid `end` from being `()`, then that is a violation. An example is presented below.

```
> (call/initial-state 1
    (call/fresh
      (lambda (x)
        (conj
          (listo x)
          (conj
            (not-pairo x)
            (disj
              (=/= '() x)
              (absento x '()))))))))
'()
```

In the last predicate required to correctly implement `listo`, `end` can be a proper list of unknown length. An `absento` constraint forbidding `()` from occurring in a term containing `end`, however, causes a violation. The constraint must precisely forbid `()` from occurring in a term containing `end` to cause the violation.

```
> (call/initial-state 1
    (call/fresh
      (lambda (x)
        (conj
          (listo x)
          (absento '() x)))))
'()
```

These constraint-violation predicates are somewhat involved. But this is of necessity. Defining constraints requires domain-specific knowledge on the part of the constraint architect. In any implementation of constraints, information of this complexity must be included in the system, and its exact nature changes based on the constraints involved. We have ensured that constraint violations can each be treated independently and that they comprise the entirety of the constraint domain knowledge required. Furthermore, by requiring that our solver be monotonic and logical, we have ensured that adding new constraints can never require modifying old predicates.

6. Related work

miniKanren is a family of related languages with an overlapping set of operators and a common design philosophy. The seminal implementation, also named "miniKanren", was first presented in *The Reasoned Schemer*, and since then there has been a great profusion of miniKanren languages. These have included both additional constraints and control operators.

As the "mini-" and "micro-" modifiers suggest, there was an earlier language Kanren [9]. Kanren, from the Japanese meaning "relation", is also a programming language based on relation composition and relation extension in the way many functional languages are based on the extension and composition of functions. miniKanren is named with respect to the earlier Kanren, but the languages have distinct syntax, semantics, and design goals. miniKanren is "mini-" in the sense that as a language it makes more demands of the users and less automation on the part of the implementation.

There exists a close connection between microKanren (and thus also miniKanren) and a subset of Prolog. Modulo differences in syntax, a microKanren relation is essentially a *completed predicate*, à la Clark [6]. Spivey and Seres's work on a Haskell embedding of Prolog [31], Kiselyov's "Taste of Logic Programming" [20], Kiselyov et. al's Logic monad [21], and of course Ralf Hinze's extensive work on implementations of Prolog-style backtracking [13–15] are all closely related to our microKanren as well.

CLP is, in differing senses, both an extension and a generalization of traditional logic programming. It was seen as a way to extend logic programming with other constraints [17]. Simultaneously, it makes clear that pure logic programming is but one particular instance of CLP, in which unification is itself the solver.

Programming with constraints had been investigated well before the emergence of widespread interest in constraint logic programming [2, 32]. Modern development of CLP languages begins in the mid 1980s with groups in three places: Jaffar et. al in Melbourne, Colmerauer at Marseilles, and with the ECRC in Munich [26].

Lim and Stuckey's "A constraint logic programming shell" provides a framework similar to Constraint microKanren for developing constraints [25]. Jaffar and Lassez's CLP Scheme is the theoretical model into which our Constraint microKanren fits [16].

Schrijvers et al. also separate their constraint-solving mechanism from the implementation of their search [29]. Their emphasis is on implementing different search strategies via monad transformers over basic search monads. It's not yet clear where microKanren's search sits in their framework, though this is a topic we are currently investigating.

There exists a different sort of CLP paradigm based on research in constraint satisfaction problems and constraint propagation to reduce the search space. Le Provost and Wallace [24], in their "Generalized Propagation over the CLP Scheme" describe a merger of the two models.

cKanren, an earlier miniKanren for CLP, takes a different approach than constraint microKanren, using domain restriction and constraint propagation [1]. Alvis et. al take as their primary example finite domains. cKanren returns as answers ground instances that satisfy the program's constraints. Unlike Constraint microKanren's framework, they provide constraint minimization and an answer-formatter in their implementation.

7. Conclusion

We have presented a constraint logic programming version of microKanren. We have provided a framework for implementing constraints and constraint systems. Decoupling constraints from the control flow has further clarified the implementation of the constraints framework. We have implemented customary miniKanren constraints, as well as interesting and useful new ones.

We have tried to eschew all possible sophistication in the implementation. We expect Constraint microKanren will be slower than current miniKanren implementations. It will be less efficient than state-of-the-art CLP languages. Rather than efficiency, our aim is a simple, generic framework for implementing constraints in microKanren. We also envision Constraint microKanren as a lightweight, quick prototyping tool for implementing constraint systems. Constraint architects can experiment with and test both constraints and answers that result without building or modifying a complicated and efficient dedicated solver.

Though efficiency isn't a concern, it might still be interesting to see the performance impacts of various simple optimizations. These might include incremental constraint solving, early projection [8], attributed variables [23], or calling out to a dedicated constraint solver where easily applicable.

Also future work is a generic, extensible constraint minimization routine similar to `invalid?`. The architect should be able to specify the constraints minimizations individually and allow the framework to produce a minimizer.

We also seek to establish a more rigorous definition of a constraint-violation category. Describing precisely what violations should be checked by a single predicate would better clarify the constraint architect's responsibilities. We hope also to formalize the connection of the Constraint microKanren framework to the CLP Scheme.

We believe that we achieve a simple, portable, and understandable model of CLP. Further, we believe that constraint microKanren lays the foundations for continued future work in designing constraint systems.

Acknowledgements

We thank Will Byrd, Chung-chieh Shan, and Oleg Kiselyov for early discussions of constraints in miniKanren. We thank Ryan Culpepper for his improvements to the framework macros. We also thank our anonymous reviewers for their suggestions and improvements.

References

[1] C. E. Alvis, J. J. Willcock, K. M. Carter, W. E. Byrd, and D. P. Friedman. cKanren: miniKanren with constraints. *Scheme and Functional Programming*, 2011.

[2] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):353–387, 1981.

[3] C. Brozefsky. Core.logic and SQL killed my ORM, 2013. URL `http://www.infoq.com/presentations/Core-logic-SQL-ORM`.

[4] W. E. Byrd and D. P. Friedman. αKanren: A fresh name in nominal logic programming. In *Proceedings of the 2007 Workshop on Scheme and Functional Programming, Université Laval Technical Report DIUL-RT-0701*, pages 79–90 (*see also* `http://www.cs.indiana.edu/~webyrd` *for improvements*), 2007.

[5] W. E. Byrd, E. Holk, and D. P. Friedman. miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl). In *2012 Workshop on Scheme and Functional Programming*, Sept. 2012.

[6] K. L. Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.

[7] R. Culpepper and M. Felleisen. Fortifying macros. *ACM Sigplan Notices*, 45(9):235–246, 2010.

[8] A. Fordan. *Projection in Constraint Logic Programming*. Ios Press, 1999.

[9] D. P. Friedman and O. Kiselyov. A declarative applicative logic programming system, 2005. URL `http://kanren.sourceforge.net/`.

[10] D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer*. The MIT Press, Cambridge, MA, 2005.

[11] D. Gregoire. Web testing with logic programming, 2013. URL `http://www.youtube.com/watch?v=09zlcS49zL0`.

[12] J. Hemann and D. P. Friedman. μKanren: A minimal functional core for relational programming, 2013. URL `http://schemeworkshop.org/2013/papers/HemannMuKanren2013.pdf`.

[13] R. Hinze. Deriving backtracking monad transformers. In *ACM SIGPLAN Notices*, volume 35, pages 186–197. ACM, 2000.

[14] R. Hinze. Prolog's control constructs in a functional setting axioms and implementation. *International journal of foundations of computer science*, 12(02):125–170, 2001.

[15] R. Hinze et al. Prological features in a functional setting: Axioms and implementation. In *Fuji International Symposium on Functional and Logic Programming*, pages 98–122, 1998.

[16] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 111–119, New York, NY, USA, 1987. ACM. . URL http://doi.acm.org/10.1145/41625.41635.

[17] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19:503–581, 1994.

[18] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. Yap. Projecting CLP($\mathcal{R}$) constraints. *New Generation Computing*, 11 (3-4):449–469, 1993.

[19] J. Jaffar, M. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *The Journal of Logic Programming*, 37(1):1–46, 1998.

[20] O. Kiselyov. The taste of logic programming, 2006. URL http://okmij.org/ftp/Scheme/misc.html#sokuza-kanren.

[21] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). In *ACM SIGPLAN Notices*, volume 40, pages 192–203. ACM, 2005.

[22] R. Kowalski. *Logic for problem solving*, volume 7. Ediciones Díaz de Santos, 1979.

[23] S. Le Huitouze. A new data structure for implementing extensions to Prolog. In *Programming Language Implementation and Logic Programming*, pages 136–150. Springer, 1990.

[24] T. Le Provost and M. Wallace. Generalized constraint propagation over the CLP scheme. *The Journal of Logic Programming*, 16(3):319–359, 1993.

[25] P. Lim and P. J. Stuckey. A constraint logic programming shell. In *Programming Language Implementation and Logic Programming*, pages 75–88. Springer, 1990.

[26] K. Marriott and P. J. Stuckey. *Programming with constraints: an introduction*. MIT press, 1998.

[27] miniKanren.org. minikanren-with-symbolic-constraints, 2015. URL https://github.com/webyrd/miniKanren-with-symbolic-constraints.

[28] J. P. Near, W. E. Byrd, and D. P. Friedman. αleanTAP: A declarative theorem prover for first-order classical logic. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 238–252. Springer-Verlag, Heidelberg, 2008.

[29] T. Schrijvers, P. J. Stuckey, and P. Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(06):663–697, 2009.

[30] R. Senior. Practical core.logic, 2012. URL http://www.infoq.com/presentations/core-logic.

[31] J. Spivey and S. Seres. Embedding Prolog in Haskell. In *Proceedings of Haskell Workshop*, volume 99, pages 1999–28, 1999.

[32] G. L. Steele. *The definition and implementation of a computer programming language based on constraints*. PhD thesis, Massachusetts Institute of Technology, 1980.

[33] B. A. Tate, I. Dees, F. Daoud, and J. Moffitt. *Seven More Languages in Seven Weeks: Languages That Are Shaping the Future*. Pragmatic Bookshelf, 2014.

[34] M. Wallace. Constraint logic programming. In *Computational logic: Logic programming and beyond*, pages 512–532. Springer, 2002.

[35] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2): 67–96, 2012.

Appendix: microKanren

```
#| Nat → Var |#
(define (var n) n)
#| Term → Bool |#
(define (var? n) (number? n))
#| Var × Term × Subst → Bool |#
(define (occurs? x v s)
  (let ((v (walk v s)))
    (cond
      ((var? v) (eqv? x v))
      ((pair? v) (or (occurs? x (car v) s)
                     (occurs? x (cdr v) s)))
      (else #f))))
#| Var × Term × Subst → Maybe Subst |#
(define (ext-s x v s)
  (cond
    ((occurs? x v s) #f)
    (else `((,x . ,v) . ,s))))
#| Term × Subst → Term |#
(define (walk u s)
  (let ((pr (assv u s)))
    (if pr (walk (cdr pr) s) u)))
#| Term × Term × Subst → Maybe Subst |#
(define (unify u v s)
  (let ((u (walk u s)) (v (walk v s)))
    (cond
      ((eqv? 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 #f))))
#| (Var → Goal) → State → Stream |#
(define ((call/fresh f) S/c)
  (let ((S (car S/c)) (c (cdr S/c)))
    ((f (var c)) `(,S . ,(+ 1 c)))))
#| Stream → Stream → Stream |#
(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))
#| Goal → Stream → Stream |#
(define ($append-map g $)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map g (force $))))
    (else ($append (g (car $)) ($append-map g (cdr $))))))
#| Goal → Goal → Goal |#
(define ((disj g1 g2) S/c) ($append (g1 S/c) (g2 S/c)))
#| Goal → Goal → Goal |#
(define ((conj g1 g2) S/c) ($append-map g2 (g1 S/c)))
#| Stream → Mature Stream |#
(define (pull $) (if (promise? $) (pull (force $)) $))
#| Maybe Nat⁺ × Mature → List State |#
(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car (pull $))))
    (else (cons (car $)
            (take (and n (- n 1)) (pull (cdr $)))))))
#| Maybe Nat⁺ × Goal → List State |#
(define (call/initial-state n g)
  (take n (pull (g `(,S0 . 0)))))
(define-syntax-rule (define-relation (rid . args) g)
  (define ((rid . args) S/c) (delay/name (g S/c))))
```