Temporal Logic, μKanren, and a Time-Traveling RDF Database

NATHANIEL RUDAVSKY-BRODY*, Independent Researcher, USA

By adding a temporal primitive to the μKanren language and adapting its interleaving search strategy to preserve simultaneity in linear time, we show how a system of temporal relational programming can be implemented. This system is then applied to a practical problem in distributed systems and linked data, as we describe a simple data store that can provide incremental solutions (deltas) to complex queries as new records are added to, or removed from the database.

Additional Key Words and Phrases: miniKanren, relational programming, temporal logic, RDF, microservices

1 INTRODUCTION

When using logic programming to model or interact with stateful resources, it is convenient to have a way to reason about time. To this end, temporal logic, and in particular linear temporal logic, has been formalized as a modal extension to predicate logic. Most definitions of linear temporal logic begin with two modal operators, Next (`X`) and Until (`U`), which are used to construct a larger set of operators for conveniently reasoning about time. Informally speaking, for the formulas `a` and `b` we say that `X b` holds if `b` is true in the next time step, and `a U b` holds if `a` is always true until `b` is true, and `b` is eventually true. For a good review of this work, see [Orgun and Ma 1994].

A number of implementations of linear temporal logic have been proposed as extensions to logic languages such as Prolog. This paper stems from the simple insight that an analogous modeling of linear time can be achieved in the miniKanren family of languages [Friedman et al. 2018], a shallowly embedded logic programming DSL with interleaving depth-first search. Adding a single temporal primitive and adapting the interleaving search strategy to account for the concepts of 'now' (simultaneity) and 'later' allows us to build tools for temporal reasoning that turn out to integrate quite well with the basic methods of relational programming. The new temporal primitive provides a way of controlling simultaneity and goal construction in miniKanren programs, and be extended in a number of ways that are interesting from both a theoretical and practical standpoint.

This paper is organized in three parts. In the first two sections we go through the low-level implementation, using as our starting point the minimalist μKanren language. Then, we sketch out some preliminary ideas on how this operator might be extended to support a more complete linear temporal logic. Finally, we turn to a real-world application in the domain of distributed systems and linked data, and see how the same low-level operator can be used to build a time-aware accessor to a stateful data structure leveraging miniKanren's interleaving search.[1]

The motivation for the last example is as follows. In distributed systems such as a microservice architecture[2] we often want to send push updates based on *deltas*, or entries that have been added

*Parts of the research for this paper were undertaken while the author was employed as Semantic Applications Developer at Tenforce (Leuven, Belgium). The author thanks Tenforce for its support in undertaking this research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of Tenforce.

[1]The full code for this paper is available at https://github.com/nathanielrb/time-in-microKanren.

[2]For the specific context, see [Versteden and Pauwels 2016] and [Versteden and Pauwels 2018].

Author's address: Nathaniel Rudavsky-Brody, Independent Researcher, Sebastopol, California, 95472, USA, nathaniel@quince.studio.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

to or removed from the database. In practice, this often means having a service that indiscriminately pushes all deltas to interested subscribers; it is then the responsibility of each subscriber to filter the deltas and determine which, if any, are relevant to its own operations. If we can calculate deltas to specific queries, however, this whole process can be greatly refined. Ultimately we will describe how a simple RDF database (triple store) can be implemented using temporal μKanren as its query language, that calculates deltas to specific queries via a system of incremental indexes.

2 BASIC DEFINITIONS

We begin by briefly reviewing the definitions of μKanren as described in [Hemann and Friedman 2013] and [Hemann et al. 2016], deferring to those articles for discussion and motivations. To better expose the core concepts, we choose to work with the earlier implementation[3] which has the advantage of using procedures to represent immature streams, leaving us free to use Scheme promises for time. Readers familiar with its implementation may wish to skip to the next section. Though the two versions of μKanren run to 39 and 54 lines of code respectively, here we will only repeat the definitions from [Hemann and Friedman 2013] relevant to understanding the changes that will be introduced afterwards. (The omitted definitions can be referred to in the final implementation included in Appendix A.)

A μKanren program operates by applying a *goal*, analogous to a predicate in logic programming and implemented as a procedure of one argument, to a *state*, defined as a substitution (an association list of variables and their values) paired with a variable counter. Variables are represented as vectors containing their variable index. The program can succeed or fail, and when it succeeds returns a sequence of new states, called a *stream*, each one extending the original state by new variable substitutions that make the goal succeed.

Goals are built with four basic goal constructors, ==, `call/fresh`, `disj` and `conj`, to be defined below. In the following example, a goal made of the disjunction of two == goals is applied to the empty state. The program returns a stream of two states, each corresponding to one branch of the disjunction.

```
(define empty-state '(() . 0))

((call/fresh
  (lambda (q)
    (disj (== q 4)
          (== q 5))))
 empty-state)
;; => '((((#(0) . 4)) . 1) (((#(0) . 5)) . 1))
```

The `walk` operator, not defined here, searches for a variable's value in a substitution, while substitutions are extended (without checking for circularities) by consing on the new substitution. `walk` is used by the `unify` operator, which recursively unifies two variables with respect to a given variable substitution, extending the substitution when it succeeds, and returning `#f` when it fails. It is `unify`'s behavior that defines the basic terms of μKanren as being variables, objects equivalent under `eq?`, and pairs of such terms.

The fundamental goal constructor is ==, which builds a goal that succeeds if its two arguments can be unified in the current state, and otherwise returns the empty stream (`mzero`).

[3]See https://github.com/jasonhemann/microKanren for the code.

```
(define (== u v)
  (lambda (s/c)
    (let ((s (unify u v (car s/c))))
      (if s (unit `(,s . ,(cdr s/c))) mzero))))
```

The conj and disj goal constructors return goals that succeed if, respectively, both or either of the goals passed as arguments succeed. They are defined in terms of mplus and bind, which together implement μKanren's interleaving search strategy.

```
(define (disj g1 g2) (lambda (s/c) (mplus (g1 s/c) (g2 s/c))))
(define (conj g1 g2) (lambda (s/c) (bind (g1 s/c) g2)))

(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    ((procedure? $1) (lambda () (mplus $2 ($1))))  ; interleave
    (else (cons (car $1) (mplus (cdr $1) $2)))))

(define (bind $ g)
   (cond
     ((null? $) mzero)
     ((procedure? $) (lambda () (bind ($) g)))
     (else (mplus (g (car $)) (bind (cdr $) g)))))
```

The interleaving itself, which guaranties complete search without the performance penalty of breadth-first search, is effectuated by the second case of mplus by exchanging the order of $1 and $2. Moreover, the second case in each of the two operators implements *immature streams* as lambda expressions, allowing programs to return infinite streams of results. In μKanren it is the user's responsibility to correctly handle immature streams, invoking them if necessary. A regular stream, namely a pair of a state and a stream, is correspondingly termed a *mature stream*.

3 TIME IN μKANREN

To model linear time in μKanren we begin by introducing a third type of stream, *delayed streams*, to represent goals that are delayed until a later point in time. We can represent delayed streams by promises, and construct them using a single temporal goal constructor next analogous to the Next (X) operator introduced above.

```
(define-syntax next
  (syntax-rules ()
    ((_ g) (lambda (s/c) (delay (g s/c))))))
```

A complex goal might have many levels of delays representing different future points in time. It is important that the temporal order of these subsidiary goals be preserved during interleaving search, both with regards to the ordering of solutions and also to guaranty that a goal (which might refer to a stateful resource) is constructed at the right time. We accomplish this by adapting the interleaving process through the addition of two cases to the definition of mplus, so that delayed goals (promises) are shunted right, all promises at the same nested level are recombined together. Among other properties, this preserves the order-independence of conjunction and disjunction.

```
(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    ((procedure? $1) (lambda () (mplus $2 ($1))))
        ((and (promise? $1) (promise? $2))
     (delay (mplus (force $1) (force $2))))         ; recombine
    ((promise? $1) (mplus $2 $1))                   ; shunt right
    (else (cons (car $1) (mplus (cdr $1) $2)))))
```

bind also needs to be modified by adding an additional case to delay the binding of delayed streams with a goal. The utility function forward is used to 'fast-forward' through the enclosing delayed stream.

```
(define (forward g)
  (lambda (s/c)
    (let rec (($ (g s/c)))
      (cond ((null? $) '())
            ((promise? $) (force $))
            ((procedure? $) (lambda () (rec ($))))
            (else (cons (car $) (rec (cdr $))))))))

(define (bind $ g)
   (cond
     ((null? $) mzero)
     ((procedure? $) (lambda () (bind ($) g)))
     ((promise? $) (delay (bind (force $) (forward g))))
     (else (mplus (g (car $)) (bind (cdr $) g)))))
```

The resulting 53 lines of code make up the functional core of our temporal μKanren. To keep the subsequent code simple, we extend the miniKanren wrappers from the original paper to handle delayed streams, saving the code for Appendix B. Using the miniKanren wrappers, and in particular run* instead of call/fresh to reify solutions, improves the readability of μKanren's data representation. We also introduce a few convenience functions for accessing delayed streams, here defined with the help of take-right and drop-right from SRFI-1.

```
(define (promised $)
  (take-right $ 0))

(define (current $)
  (drop-right $ 0))

(define (advance $)
  (let ((p (promised $)))
    (and p (force p))))
```

We are ready for a simple example. Here we use disj to specify that q equals *db* either now or at the next point in time. The first solution in the returned stream is none other than 1, the current value of *db*, while the promise contains the delayed stream to be advanced at a future point in time.

```
(define *db* 1)
```

```
(define r
  (run* (q)
    (fresh (a b)
      (disj (== q *db*)
            (next (== q *db*))))))
;; => (1 . #<promise>)
```

Now we can update our database and advance `r` to continue forward in time. At this point the stream is fully mature, since `next` is not used recursively.

```
(set! *db* 2)

(advance r)
;; => (2)
```

The shunt-right mechanism turns out to have interesting properties for interleaving search even without referring to a stateful resource, for instance in ordering and grouping solutions, as the following contrived example shows. We start by defining `inco` which recursively generates a 'now or later' type of goal, incrementing its variable at each future point in time.

```
(define (inco x)
  (let rec ((n 0))
    (disj (== x n)
          (next (rec (+ n 1))))))
```

The program `s` uses `inco` to generate pairs of such incrementing variables.

```
(define s
  (run* (q)
    (fresh (a b)
      (== q `(,a ,b))
      (conj (inco a) (inco b)))))
```

As we advance `s`, we see that the solutions are grouped in increasing order of the maximum value of the two variables.

```
s
;; => ((0 0) . #<promise>))

(advance s)
;; => ((0 1) (1 0) (1 1) . #<promise>))

(advance (advance s))
;; => ((1 2) (2 0) (2 1) (2 2) (0 2) . #<promise>))

(advance (advance (advance s)))
;; => ((0 3) (2 3) (3 0) (3 1) (3 2) (3 3) (1 3) . #<promise>))
```

Indeed, this points up the difference between two basic uses, or two interpretations, of temporal µKanren. In the first case, when interacting with a stateful resource, we are doing logic programming *in* time. When reasoning *about* time in a truly relational way, on the other hand, we are more interested in the combinatoric properties of `next`. We will scratch the surface of this distinction in the next section, but a deeper consideration of its implications remains to be done.

4 TOWARDS A TEMPORAL RELATIONAL PROGRAMMING

As we saw in the introduction, linear temporal logic extends predicate logic with temporal modal operators referring to linear time. Several of these operators can be usefully recovered in temporal µKanren using the `next` primitive defined above. We will not develop a formally complete system here, but rather provide some heuristic examples to suggest how such a system might developed, noting along the way its potential limitations and specificities.

Most temporal operators need to be defined recursively. When doing this, it is essential to control when the goal is constructed, since the goal might refer to a stateful resource. For temporal µKanren we can do this by wrapping goals in a lambda expression which are evaluated at the appropriate time. The definition of `precedes`, modeled on Weak Until (`W`) and which stipulates that a goal `g` holds at least until a second goal `h` holds, shows how this can be done.

```
(define (precedes* g* h*)
  (let ((g (g*)) (h (h*)))
    (disj h (conj g (next (precedes* g* h*))))))

(define-syntax precedes
  (syntax-rules ()
    ((_ g h) (precedes* (lambda () g) (lambda () h)))))
```

Translating another basic operator, Always (`G`), reveals one of the pitfalls of naively mapping temporal logic to relational programming. Our first impulse is to define it analogously to `until`.

```
(define (always* g*)
  (let ((g (g*)))
    (conj g (next (always* g*)))))

(define-syntax always
  (syntax-rules ()
    ((_ g) (always* (lambda () g)))))
```

A little experimentation, however, shows how this definition can be problematic. A goal constructed with `always` will return a promise so long as the goal it encloses holds, and the empty stream as soon as it fails. This behavior can be useful for a program that wants to verify whether a state still holds, but `always` will never construct a mature stream of solutions, since clearly the goal will never have been true *forever*, at least not in linear time.

There are two ways of addressing this limitation, depending on how we want to use our operator. One is to modify the definition of `next` in such a way as to allow for an `eot` (end-of-time) accessor that would essentially cut off the forward recursion of delayed promises. Another approach is to define the weaker and impure `as-long-as`, which constructs a stream of solutions to the goal `h` that continues as long as the goal `g` still holds. This technique of writing 'guarded' goal constructors is reminiscent of the implementation of cuts in miniKanren.

```
(define (as-long-as* g* h*)
  (let ((g (g*)) (h (h*)))
    (lambda (s/c)
      (let (($ (g s/c)))
        (if (null? $) mzero
            (bind $ (disj h (next (as-long-as* g* h*)))))))))
```

```
(define-syntax as-long-as
  (syntax-rules ()
    ((_ g h) (as-long-as* (lambda () g) (lambda () h)))))
```

The Eventually (F) operator also has many useful applications, but presents a similar caveat regarding infinite time. Still, the following naive definition will be useful in some contexts.

```
(define (eventually* g*)
  (let ((g (g*)))
    (disj g (next (eventually* g*)))))

(define-syntax eventually
  (syntax-rules ()
    ((_ g) (eventually* (lambda () g)))))
```

The guarded definition, however, provides more intuitive behavior, by cutting off further recursion when the goal produces a non-empty stream of solutions.

```
(define (eventually* g*)
  (let ((g (g*)))
    (lambda (s/c)
      (let (($ (g s/c)))
        (if (null? $)
        (bind (list s/c) (next (eventually* g*)))
        $)))))

(define-syntax eventually
  (syntax-rules ()
    ((_ g) (eventually* (lambda () g)))))
```

Either one can be used to define an analogue of strong Until, which along with Next is one of the building blocks of linear temporal logic.

```
(define-syntax until
  (syntax-rules ()
    ((_ g h) (conj (precedes g h) (eventually h)))))
```

Clearly this is only a beginning, and as the above examples suggest, what constitutes a *useful* temporal relational programming language will be determined in part by the application domain. Still, developing a formally satisfying version of such a system would be an obvious next step. All this side-stepping of infinite time also begs the question of what it really means to model it, a question we will not explore further here, since our primary focus is on using miniKanren to interact with stateful resources in a temporally-aware way. Giving a satisfactory answer to this question could be a fruitful direction for further work.

Another would be recovering temporal μKanren's functionality in a version of miniKanren with constraints, namely cKanren [Alvis et al. 2011] or the extended μKanren described in [Hemann and Friedman 2015]. It would also be interesting to explore the integration of temporal logic with different representations of negation in miniKanren that have been pursued in both the published and unpublished literature.

5 CALCULATING DELTAS WITH INCREMENTAL SEARCH

Now we turn to the practical application described in the introduction, and see how our temporal primitive `next` can be used to implement a simple data store with temporally-aware incremental search. We begin by describing the implementation, and then present the full code in the following section.

Our goal is to design an RDF database (triple store) using temporal µKanren as its query language, that calculates new or invalidated solutions to specific queries as records are updated in the database. By using the `next` constructor and a simple system of incremental indexes, we can store the final search positions for a query. Running a query will return both the current results and a delayed stream that when advanced will continue searching at the previous search-tree's leaves. Therefore, advancing the delayed stream after the database has been updated will return solutions that have been added to, or subtracted from the solution set.

An RDF database [Manola and Miller 2004] stores semantic facts, or triples,[4] made up of a *subject* (a URI), a *predicate* (a URI), and an *object* (a URI, or a string, boolean or numeric literal).

```
<http://ex.org/people/1> <http://xmlns.com/foaf/0.1/name> "John"
```

A minimalist triple store can be implemented using three indexes, allowing for the retrieval of groups of triples matching a given query pattern: `spo`, `pos`, and `osp`, where `s` is the subject, `p` the predicate, and `o` the object. For the query pattern `?s <P> <O>`, for instance, `?s` is our only variable so the database would use the `pos` index for fastest lookup. Here, however, we want to know *incremental indexes*: for a given `s`, we need the indexed `p`s in a form that can be easily compared to future `p`s to determine whether new keys have been added.

Our database is therefore defined as a set of seven incremental indexes (`null`, `s`, `sp`, `p`, `po`, `o`, `os`) plus the `spo` index for full triples. Adding a triple to the database is a matter of `cons`ing each element to the appropriate incrementals lists and adding a truthy value (here `#t`, though this could also be a more meaningful identifier or timestamp) to the full `spo` index. When deleting a triple, we leave the index keys, but update the triple's value in `spo` to `#f`. Here are the incremental indexes after adding the triple `<A> <B> <C>`.

```
<ø   [ #f => (<A> ...) ] >
<s   [ <A> => (<B> ...) ] >
<sp  [ (<A> <B>) => (<C> ...) ] >
<p   [ <B> => (<C> ...) ] >
<po  [ (<B> <C>) => (<A> ...) ] >
<o   [ <C> => (<A> ...) ] >
<os  [ (<C> <A>) => (<B> ...) ] >
<spo [ (<A> <B> <C>) => #t ] >
```

The main accessor is the goal constructor `triple-nolo` ('triple-now-or-later') that descends recursively through the incremental indexes one level at a time, constructing a stream with the current indexes and saving the last search positions in a delayed stream. A dynamic parameter is used to specify the current database state.

To see our data store in action, we consider the following query written in SPARQL 1.1 [Harris and Seaborne 2013], the standard query language for RDF data stores. This query will be translated into temporal µKanren and run against successive states of the database.

```
SELECT ?o
WHERE {
  <S> <P> ?o.
  <Q> <R> ?o.
}
```

First, however, we need to define an empty database.

[4]Here we will only consider triples, though triple stores actually store quads, with the addition of the *graph* element.

```
(define db0 (empty-db))
```

Then the SPARQL query is translated into the temporal μKanren goal `r` using `triple-nolo`. To make it clear what is going on, we keep track of the individual delta flags for each triple goal, though in practice we usually want to know only whether the solution was added (all +s) or removed (at least one `-`). Since there are no solutions to our query, the program returns a delayed stream.

```
(define r
  (parameterize ((latest-db db0))
    (run* (q)
      (fresh (o deltas d1 d2)
        (== q `(,deltas  ,o))
        (== deltas `(,d1 ,d2))
        (triple-nolo  d1 '<S> '<P> o)
        (triple-nolo  d2 '<Q> '<R> o)))))
;; => #<promise>
```

Now we can add some triples and advance the delayed stream.

```
(define db1
  (add-triples db0  '((<S> <P> <O1>)
                      (<S> <P> <O2>)
                      (<Q> <R> <O1>)
                      (<A> <B> <C>))))

(parameterize ((latest-db db1))
  (advance r))
;; => (((+ +) <O1>) . #<promise>)
```

If we delete a triple that contributed to one of our solutions, that solution will be returned with a negative delta flag in the next iteration.

```
(define db2
  (delete-triples db1 '((<S> <P> <O1>))))

(parameterize ((latest-db db2))
  (advance (advance r)))
;; => (((+ -)) <O1>) . #<promise>)
```

Finally we add that triple back along with some new solutions, to get positive deltas.

```
(define db3
  (add-triples db2 '((<S> <P> <O1>)
                     (<S> <P> <O3>)
                     (<Q> <R> <O3>)
                     (<S> <P> <M>)
                     (<Q> <R> <M>))))

(parameterize ((latest-db db3))
  (advance (advance (advance r))))
;; => (((+ +) <O1>) ((+ +) <M>) ((+ +) <O3>) . #<promise>)
```

Here we have been proceeding linearly, but since the database states are persistent we can calculate deltas over any two states. Keeping track of a stream of states and indexing them on time will therefore give us a truly time-traveling data store.

6 IMPLEMENTING THE RDF STORE

Now we are ready to walk through the full code for the data store.

We start by defining `project`, a standard miniKanren operator [Byrd 2009] that binds a set of variables to their current substitutions. It will be used control `triple-nolo`'s recursive descent through the indexes.

```
(define-syntax project
  (syntax-rules ()
    ((_ (x ...) g)
     (lambda (s/c)
       (let ((x (walk x (car s/c))) ...)
         (g s/c))))))
```

A database is defined as a set of seven incremental indexes plus the full spo index, as described above. We keep a separate incremental index for each index level, storing the incrementals as a simple cons list along with a map for quick existence checking. For each index itself we use a hash array mapped trie[5] that allows for persistently storing successive states of the database through path copying.

```
(define-record db null s sp p po o os spo)

(define-record incrementals map list)

(define (empty-incrementals)
  (make-incrementals (persistent-map) '()))

(define (empty-db)
  (apply make-db
         (make-list 8 (persistent-map))))
```

Dynamic parameters are used to specify the current database state.

```
(define latest-db (make-parameter (empty-db)))

(define (latest-incrementals accessor key)
  (incrementals-list
   (map-ref (accessor (latest-db)) key (empty-incrementals))))

(define (latest-triple s p o)
  (map-ref (db-spo (latest-db))
           (list s p o)))
```

To update the indexes we update the quick-check map to `#t` and cons the new value to the incrementals list.

[5]In the Chicken implementation presented here, we use Persistent Hash Maps [Heidkamp 2013], derived from Ian Price's Hash Array Mapped Tries [Price 2014].

```
(define (update-incrementals table key val)
  (let ((incrementals (map-ref table key (empty-incrementals))))
    (map-add table key
             (if (map-ref (incrementals-map incrementals) val)
                 incrementals
                 (make-incrementals
                  (map-add (incrementals-map incrementals) val #t)
                  (cons val (incrementals-list incrementals)))))))

(define (update-triple table triple val)
  (map-add table triple val))
```

Updating (adding or removing) a triple is therefore a matter of updating all eight indexes.

```
(define (update-triples DB triples val)
  (let loop ((triples triples)
             (i/null (db-null DB))
             (i/s (db-s DB))
             (i/sp (db-sp DB))
             (i/p (db-p DB))
             (i/po (db-po DB))
             (i/o (db-o DB))
             (i/os (db-os DB))
             (i/spo (db-spo DB)))
    (if (null? triples)
        (make-db i/null i/s i/sp i/p i/po i/o i/os i/spo)
        (match (car triples)
          ((s p o)
           (loop (cdr triples)
                 (update-incrementals i/null #f s)
                 (update-incrementals i/s s p)
                 (update-incrementals i/sp (list s p) o)
                 (update-incrementals i/p p o)
                 (update-incrementals i/po (list p o) s)
                 (update-incrementals i/o o s)
                 (update-incrementals i/os (list o s) p)
                 (update-triple i/spo (list s p o) val)))))))

(define (add-triples DB triples) (update-triples DB triples #t))

(define (delete-triples DB triples) (update-triples DB triples #f))

(define (add-triple s p o)
  (add-triples `((,s ,p ,o))))

(define (delete-triple s p o)
  (delete-triples `((,s ,p ,o))))
```

At last we can define our main accessor, `triple-nolo`. This operator descends recursively through the indexes, selecting which index to use according to which of its arguments are variables (var?) and using project in the recursion step to reduce the number of variables by one. The stream of solutions is constructed by stream that recurses through the incrementals list (indexes), starting at the head and stopping when it reaches the previous head (ref), and storing the new head of the list in another recursive call wrapped in next as the delayed stream of future solutions.

```
(define (triple-nolo delta s p o)
  (let ((mkstrm (lambda (var accessor key)
                  (let* ((get-incrementals (lambda ()
                                             (latest-incrementals accessor key)))
                         (initial-indexes (get-incrementals)))
                    (let stream ((indexes initial-indexes)
                                 (ref '())
                                 (next-ref initial-indexes))
                      (if (equal? indexes ref)
                          (next
                           (let ((vals (get-incrementals)))
                             (stream vals next-ref vals)))
                          (disj
                           (conj (== var (car indexes))
                                 (project (s p o)
                                   (triple-nolo delta s p o)))
                           (stream (cdr indexes) ref next-ref))))))))
    (cond ((and (var? s) (var? p) (var? o)) (mkstrm s db-null #f))
          ((and (var? s) (var? p))          (mkstrm s db-o o))
          ((and (var? s) (var? o))          (mkstrm o db-p p))
          ((and (var? p) (var? o))          (mkstrm p db-s s))
          ((var? s)                         (mkstrm s db-po (list p o)))
          ((var? p)                         (mkstrm p db-os (list o s)))
          ((var? o)                         (mkstrm o db-sp (list s p)))
          (else
           (let leaf ((ref #f))
             (let ((v (latest-triple s p o)))
               (cond ((eq? v ref) (next (leaf v)))
                     (v           (disj (== delta '+) (next (leaf v))))
                     (else        (disj (== delta '-) (next (leaf v)))))))))))
```

Though this reduced example is clearly far from a production-ready system, we hope we have demonstrated one of the practical uses of temporal relational programming. Indeed, we intend to pursue these tools in an industrial setting, and the full implementation will be a laboratory for other practical applications of miniKanren, such as drawing on [Byrd et al. 2012] to compile SPARQL to miniKanren and using search ordering as explored in [Swords and Friedman 2013] to aid with query optimization.

A APPENDIX: TEMPORAL μKANREN

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

(define (walk u s)
  (let ((pr (and (var? u) (assp (lambda (v) (var=? u v)) s))))
    (if pr (walk (cdr pr) s) u)))

(define (ext-s x v s) `((,x . ,v) . ,s))

(define (== u v)
  (lambda (s/c)
    (let ((s (unify u v (car s/c))))
      (if s (unit `(,s . ,(cdr s/c))) mzero))))

(define (unit s/c) (cons s/c mzero))
(define mzero '())

(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 (call/fresh f)
  (lambda (s/c)
    (let ((c (cdr s/c)))
      ((f (var c)) `(,(car s/c) . ,(+ c 1))))))

(define (disj g1 g2) (lambda (s/c) (mplus (g1 s/c) (g2 s/c))))
(define (conj g1 g2) (lambda (s/c) (bind (g1 s/c) g2)))

(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    ((procedure? $1) (lambda () (mplus $2 ($1))))
    ((and (promise? $1) (promise? $2))
     (delay (mplus (force $1) (force $2))))
    ((promise? $1) (mplus $2 $1))
    (else (cons (car $1) (mplus (cdr $1) $2)))))

(define (forward g)
  (lambda (s/c)
    (let rec (($ (g s/c)))
      (cond ((null? $) '())
        ((promise? $) (force $))
            ((procedure? $) (lambda () (rec ($))))
        (else (cons (car $) (rec (cdr $))))))))
```

```
(define (bind $ g)
   (cond
     ((null? $) mzero)
     ((procedure? $) (lambda () (bind ($) g)))
     ((promise? $) (delay (bind (force $) (forward g))))
     (else (mplus (g (car $)) (bind (cdr $) g)))))

(define-syntax next
  (syntax-rules ()
    ((_ g) (lambda (s/c) (delay (g s/c))))))
```

B APPENDIX: MINIKANREN WRAPPERS

Here we adapt the miniKanren control operators described in [Hemann and Friedman 2013] for delayed streams. The main changes are to the definitions of `run`, `run*`, `pull`, `take-all` and `take`. Unlike the original paper, we do not use `Zzz` to wrap goals in `conj+` and `disj+`, since this makes it difficult to control the goal construction time at different levels of nested delays, a problem when referring to stateful resources. This difficulty should be addressed in future implementations of temporal μKanren.

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

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

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

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

(define-syntax run
  (syntax-rules ()
    ((_ n (x ...) g0 g ...)
     (let r ((k n) ($ (take n (call/goal (fresh (x ...) g0 g ...)))))
       (cond ((null? $) '())
         ((promise? $) (delay (r (- k 1) (take k (force $)))))
         (else (cons (reify-1st (car $))
            (r (- k 1) (cdr $))))))))

(define-syntax run*
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (let r (($ (take-all (call/goal (fresh (x ...) g0 g ...)))))
       (cond ((null? $) '())
         ((promise? $) (delay (r (take-all (force $)))))
         (else (cons (reify-1st (car $))
             (r (cdr $)))))))))

(define empty-state '(() . 0))

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

(define (pull $)
  (cond ((procedure? $) (pull ($)))
    ((promise? $) $)
    (else $)))

(define (take-all $)
  (let (($ (pull $)))
    (cond ((null? $) '())
      ((promise? $) $)
      (else (cons (car $) (take-all (cdr $)))))))

(define (take n $)
  (if (zero? n) '()
    (let (($ (pull $)))
      (cond ((null? $) '())
        ((promise? $) $)
        (else (cons (car $) (take (- n 1) (cdr $))))))))

(define (reify-1st s/c)
  (let ((v (walk* (var 0) (car s/c))))
    (walk* v (reify-s v '()))))

(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 (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))))

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

(define (fresh/nf n f)
  (letrec
    ((app-f/v*
       (lambda (n v*)
         (cond
           ((zero? n) (apply f (reverse v*)))
           (else (call/fresh
                   (lambda (x)
                     (app-f/v* (- n 1) (cons x v*)))))))))
     (app-f/v* n '())))
```

REFERENCES

Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. 2011. cKanren: miniKanren with Constraints. *Proceedings of the 2011 Workshop on Scheme and Functional Programming*.

William E. Byrd. 2009. *Relational programming in miniKanren: techniques, applications, and implementations*. Ph.D. Dissertation. Bloomington, IN, USA.

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 (Scheme '12)*. ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105

Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer, Second Edition*. The MIT Press, Cambridge, MA, USA.

Steve Harris and Andy Seaborne. 2013. *SPARQL 1.1 Query Language*. Technical Report. https://www.w3.org/TR/sparql11-query/

Moritz Heidkamp. 2013. persistent-hash-map. (2013). http://wiki.call-cc.org/eggref/4/persistent-hash-map

Jason Hemann and Daniel P. Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. *Proceedings of the 2013 Workshop on Scheme and Functional Programming*.

Jason Hemann and Daniel P Friedman. 2015. A Framework for Extending microKanren with Constraints. *Proceedings of the 2015 Workshop on Scheme and Functional Programming*.

Jason Hemann, Daniel P. Friedman, William E. Byrd, and Might Matthew. 2016. A Small Embedding of Logic Programming with a Simple Complete Search. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016)*. ACM, New York, NY, USA, 96–107. https://doi.org/10.1145/2989225.2989230

Frank Manola and Eric Miller. 2004. *RDF Primer*. Technical Report. https://www.w3.org/TR/2004/REC-rdf-primer-20040210/

Mehmet A. Orgun and Wanli Ma. 1994. An Overview of Temporal and Modal Logic Programming. In *Proc. First Int. Conf. on Temporal Logic - LNAI 827*. Springer-Verlag, 445–479.

Ian A. Price. 2014. Purely Functional Data Structures in Scheme. (2014). https://github.com/ijp/pfds/blob/master/hamts.sls

Cameron Swords and Daniel Friedman. 2013. rKanren: Guided search in miniKanren. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming*.

Aad Versteden and Erika Pauwels. 2016. State-of-the-art Web Applications using Microservices and Linked Data. In *Proceedings of the 4th Workshop on Services and Applications over Linked APIs and Data*.

Aad Versteden and Erika Pauwels. 2018. mu.semte.ch: State-of-the-art Web Applications fuelled by Linked Data. (2018). Presentation at the Extended Semantic Web Conference 2018, Heraklion, Greece.