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

Nathaniel Rudavsky-Brody

Introduction

When using logical or relational programming to model or interact with stateful resources, it is convenient to have a way to reason about time. Adding a temporal primitive to the miniKanren family of languages and adapting the interleaving search 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. In addition to providing a way of controlling simultaneity and goal construction in miniKanren programs, they can be used to construct specific time-aware accessors to stateful data structures that leverage miniKanren's interleaving search.

In the first two sections we show how such a system might be implemented, and sketch out some ideas on how it can be extended to support a full linear temporal logic. For clarity, we use µKanren[1] and in particular the original implementation[2] which has the advantage of representing immature streams by procedures, leaving us free to use Scheme promises for time. In what follows, we assume a familiarity with its implementation, and only note modifications to the original code. The full code is presented in the Appendixes.

Time in µKanren

µKanren defines two types of streams, mature and immature. A third type, delayed streams, is introduced to represent goals that are delayed until a later point in time. We can represent delayed streams by Scheme promises, and construct them using a single temporal primitive `next`. (In a miniKanren where promises are used for immature streams, a custom record type can be used.)

[1] Jason Hemann Daniel P. Friedman. µKanren: A Minimal Functional Core for Relational Programming.

[2] https://github.com/jasonhemann/microKanren

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

A complex program might have many levels of delays, representing different future points in time. It is important that the temporal order of these 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 search so that promises are shunted right, and all promises at the same nested level are recombined. This is done by adding two cases to the definition of `mplus`.

```
(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)))))
```

`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)))))
```

Finally, we define a few convenience functions for accessing delayed streams.

```
(use srfi-1)

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

(define (current $)
```

```
  (drop-right $ 0))

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

Now we are ready for a simple example that demonstrates the internals of this resulting temporal µKanren.

```
(define *db* 1)

(define r
 ((call/fresh
  (lambda (q)
   (disj (== q *db*)
         (next (== q *db*)))))
  empty-state))
;; => ((((#(0) . 1)) . 1) . #<promise>)

(set! *db* 2)

(advance r)
;; => ((((#(0) . 2)) . 1))
```

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. To keep the code simple, we use the miniKanren wrappers from the original paper, extended to handle delayed streams as shown in Appendix B.

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

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

(current s)
;; => ((0 0)))

(current (advance s))
;; => ((0 1) (1 0) (1 1)))

(current (advance (advance s)))
;; => ((1 2) (2 0) (2 1) (2 2) (0 2)))
```

```
(current (advance (advance (advance  s))))
;; => ((0 3) (2 3) (3 0) (3 1) (3 2) (3 3) (1 3)))
```

Temporal Relational Programming

Linear temporal logics are generally defined in terms of two fundamental modal operators, Next and Until, that are used to define a larger set of operators for conveniently reasoning about time. We can use the `next` primitive to recover several of these operators in our temporal µKanren. We will not develop a complete system of temporal relational programming, but rather provide some notes on possible implementations and suggest directions that might be interestingly pursued.

Other than Next, most of the temporal operators need to be defined recursively, and it is essential to control when the goal is constructed. For the µKanren implementation that is our starting point here, we can do this by simply wrapping goals in a lambda expression. The definition of `until`, which stipulates that a goal `g` holds at least until a second goal `h` holds, shows how this can be done.

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

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

Implementing another basic operator, Always, reveals one of the pitfalls of mapping temporal logic to µKanren. 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) (until* (lambda () g)))))
```

A little experimentation, however, shows how this definition is 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. While this behavior can be useful for a program that only wants to verify that a state still holds, `always` will never construct a mature stream of solutions, since clearly the goal will never have been true forever.

There are two ways of getting around this kind of problem. One is to modify the definition of `next` in such a way as to allow for an `eot` (end-of-time) accessor that essentially cuts off the forward recursion of delayed promises. Another simpler approach is to define the weaker and impure `as-long-as`, the first of a series of 'guarded' goal-constructors.

```
(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 operator is also very useful, but presents a similar caveat regarding infinite time. Depending on the application, a guarded definition might be more appropriate. Here is a definition that will be useful in many contexts.

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

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

We can use it to define a version of a 'strong' Precedes.

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

Calculating Deltas With Incremental Search

Now we turn to a practical application of temporal μKanren, seeing how it can be used to implement a simple data store with temporally-aware incremental search.

The motivation is as follows. In distributed systems such as a microservice architecture, we often want to send push updates based on *deltas*, or entries that have been added to or removed from the database. In practice, this often means having a service that indiscriminately pushes all deltas to all interested subscribers, in which case it is 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 refined. Here we see how an RDF database (triple store)[3] can be implemented using temporal µKanren as its query language, that calculates deltas. 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.

A triple store can be implemented using three indexes for a triple `s p o`: `spo`, `pos`, and `osp`. Here we want to know *incremental indexes*: for a given `s`, we need a list of the available `p`'s in a form that can be easily compared to future `p`'s to determine if new keys have been added. We use hash array mapped tries[4] (persistent hash maps[5] in Chicken) that allow us to persistently store successive states of the database through path copying.

Since these hamts do not accumulate their indexes in a easily comparable way, we keep separate incremental indexes for each index level, storing the incremental indexes as simple `cons` lists along with a map for simple checking. Our database is therefore defined as a set of six incremental indexes (`s`, `sp`, `p`, `po`, `o`, `os`) plus the `spo` index for full triples. Adding a triple `<s> <p> <o>` 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 indexes, but update the triple's value in `spo` to `#f`. (Instead of keeping a null incrementals index for `s`, we overload the `s` index.)

The main accessor is the recursive goal constructor `triple-nolo` ('now or later') that descends recursively through the incremental indexes one level at a time, constructing an immature stream with the current indexes and returning the last search positions in a delayed stream. A dynamic parameter keeps track of the current database state. The full code is presented in Appendix C.

The following example shows how `triple-nolo` is used. Here we keep track of the individual deltas for each triple goal, though in practice we usually want to know only if the solution was added (all `+`s) or removed (at least one `-`).

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

(define r
  (parameterize ((latest-db db0))
    (run* (q)
          (fresh (o deltas d1 d2)
```

[3]Though RDF actually deals with quads (triple plus a graph), for simplicity we will only deal with triples here.

[4]https://github.com/ijp/pfds/blob/master/hamts.sls

[5]http://wiki.call-cc.org/eggref/4/persistent-hash-map

```
                (== q `(,deltas  ,o))
                (== deltas `(,d1 ,d2))
                (triple-nolo  d1 '<S> '<P> o)
                (triple-nolo  d2 '<Q> '<R> o)))))
;; => #<promise>

(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>)

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

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

(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>)
```

Note that since the database states are persistent, we can calculate deltas over any two states. Keeping track of a stream of states and adding an index on time will give us a truly time-traveling database.

Appendix A: 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)))
```

Appendix B: miniKanren Wrappers

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

(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 '())))
```

Appendix C: RDF Store With Incremental Indexes

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

(define-record db 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 7 (persistent-map))))

(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))

(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)))

(define (update-triples DB triples val)
  (let loop ((triples triples)
```

```
              (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/s i/sp i/p i/po i/o i/os i/spo)
        (match (car triples)
          ((s p o)
           (loop (cdr triples)
                 (update-incrementals
                  (update-incrementals i/s #f 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))))

(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 (delta 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-s #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))))))))))
```