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

Nathaniel Rudavsky-Brody

Abstract

By adding a temporal primitive to the µKanren language and adapting its interleaving search 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.

Introduction

When using logical 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, and a number of implementations have been proposed as extensions to logical languages such as Prolog.

The same thing can be achieved in the miniKanren family of relational (logic) programming languages. Adding a temporal primitive 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 time-aware accessors to stateful data structures leveraging miniKanren's interleaving search.

In the first two sections we show how such a system can be implemented, and sketch out some ideas on how it might be extended to support a more complete linear temporal logic. In the final section, we turn to a real-world application in the domain of distributed systems and linked data. For clarity, we use the minimalist µKanren language [1] and in particular the original implementation [2] which has the advantage of using procedures to represent immature streams, leaving us free to use Scheme promises for time.[1]

[1] Full code for this paper is available at https://github.com/nathanielrb/temporal-microKanren

Basic Definitions

We begin by briefly reviewing the definitions of µKanren as described in Hemann and Friedman [1], deferring to that article for discussion and motivations. Readers familier with its implementation may wish to skip to the next section.

A µKanren program operates by applying a *goal* to a *state*, defined as a substitution (association list of variables and their values) paired with a variable counter. The program can succeed or fail, and when it succeeds returns a sequence of states, or *stream*, each new state 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`. The following example uses the first three goal constructors, which will be defined below.

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

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

Variables are defined as vectors containing their variable index. The `walk` operator searches for a variable's value in a substitution, while substitutions are extended (without checking for circularities) by `ext-s`. `mzero` is the empty stream, and the `unit` operator returns a stream containing its argument as the only state.

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

The goal constructor `==` builds a goal that succeeds if its two arguments can be unified in the current state, and otherwise returns `mzero`. It depends on the `unify` operator, which defines the basic terms of µKanren as being variables, objects equivalent under `eqv?`, and pairs of such terms.

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

The goal constructor `call/fresh` allows us to bind a new logic variable and increment the variable counter.

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

Finally, the `conj` and `disj` goal constructors return that 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 implements 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))))
    (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 second case in each of the two preceding definitions 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.*

Time in µKanren

To model linear time 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 primitive `next`.

```
(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 together. 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))          ; recombine
     (delay (mplus (force $1) (force $2))))
    ((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 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))))
```

Here is a simple example that demonstrates its basic operation.

```
(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. Here and in what follows, will use the miniKanren wrappers from the original paper, likewise extended to handle delayed streams (see Appendix A).

```
(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)))
```

Towards a Temporal Relational Programming

Linear temporal logic extends predicate logic with temporal modal operators referring to linear time. Two operators, Next (`X`) and Until (`U`), are generally defined as primitives and then used to construct a larger set of operators for conveniently reasoning about time. Several of these operators can be usefully recovered in temporal µKanren using the `next` primitive. We will not develop a formally complete system here, but rather provide some heuristic examples to suggest how such a system might developed.

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 is done.

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

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

Translating another basic operator, Always (`G`), reveals one of the pitfalls of 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 alwayso
```

```
  (syntax-rules ()
    ((_ g) (always* (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 whether a state still holds, `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 getting around this problem. 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. Depending on the application, a guarded definition might be more appropriate, but the following definition will still 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 strong Until.

```
(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. Developing a complete version of such a system would be an obvious next step. Another direction to pursue would be recovering temporal µKanren's functionality in a version of miniKanren with constraints, namely cKanren [cKanren] or the extended µKanren described in [emk]. It would also be interesting to explore the integration of temporal logic with different representations of negation in miniKanren.

Calculating Deltas With Incremental Search

Now we turn to a practical application, and see how temporal µKanren can be used to implement a simple data store with temporally-aware incremental search. We proceed by describing the implementation first, and then present the full code at the end of the section.

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 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. Here we describe how an RDF database (triple store) 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.

An RDF database stores semantic facts, or triples,[2] made up of a *subject* (a URI), *predicate* (a URI), and *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. 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. We keep separate incremental indexes for each index level, storing the incrementals as a simple `cons` list along with a map for quick

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

existence checking. For each index we use a hash array mapped trie [hamt] (persistent hash maps in Chicken) that allows for persistently storing successive states of the database through path copying.

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 indexes, but update the triple's value in `spo` to `#f`. Here are the incremental indexes after adding the triple `<A> <B> <C>`.

```
<∅   [ #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.

A simple example will show the use of `triple-nolo`. We want solutions and deltas to the following query in SPARQL, the standard query language of RDF data stores.

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

We start by making our empty database.

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

The SPARQL query is translated into temporal µKanren using `triple-nolo`. To make 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 if 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 can add that triple back, along with some new solutions, to get some 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>)
```

Moreover 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.

Here is the full code.

```
(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 8 (persistent-map))))

(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-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 (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))))

(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-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 the one of the practical uses of temporal relational programming. Indeed, we intend to pursue these tools in an industrial setting [mu], and the full implementation will be a laboratory for other practical applications of miniKanren as well, notably drawing on [quines] to compile SPARQL to miniKanren and using search ordering [order] for query optimization.

Appendix A: miniKanren Wrappers

Here are the miniKanren control operators described in [1], adapted for delayed streams. Unlike the original paper, we do not use `Zzz` to wrap goals in `conj+` and `disj+` since this makes it difficult of control the goal construction time at different levels of nested delays, a problem when referring to stateful resources. This difficulty should be addressed in a more serious implementation of temporal µKanren.

```
(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 '())))
```

References

[1] J. Hemann, D. P. Friedman. µKanren: A Minimal Functional Core for Relational Programming

[2] J. Hemann and D. P. Friedman. microKanren, 2014. URL https://github.

com/jasonhemann/microKanren

[emk] J. Hemann and D. P. Friedman. A Framework for Extending microKanren with Constraints. 24th International Workshop on Functional and (Constraint) Logic Programming.

[ck] Claire E. Alvis Jeremiah J. Willcock Kyle M. Carter William E. Byrd Daniel P. Friedman. cKanren: miniKanren with Constraints. Scheme '11, Portland Oregon.

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

[mu] mu.semte.ch

[order]