DEMO: Recognizing Regular Patterns in Mixed-Type Sequences using Symbolic Finite Automata

Jim Newton

May 25, 2025

Matching an RTE

How does matching a sequence against an RTE work?
Let's first define a sequence of numbers to reuse several times.
We use `rte/match` to check it is a sequence of `Number`, *e.g.*, `(:* Number)`.

```
1 (def demo-seq [1 2 4/3 4 5 6.5 3 2])
2
3 (rte/match '(:* Number)
4            demo-seq) ;; --> true
5
6 (instance? Number 10) ;; --> true
```

Matching an RTE

Let's see whether it is a sequence of numbers which contains a `Ratio`?
What about contains a `Double`?

```
1 (def demo-seq [1 2 4/3 4 5 6.5 3 2])
2
3 (rte/match '(:cat (:* Number) Ratio (:* Number))
4            demo-seq) ;; --> true
5
6 (rte/match '(:cat (:* Number) Double (:* Number))
7            demo-seq) ;; --> true
```

Now we check whether the sequence of numbers contains both a `Double` and a `Ratio`. There are two ways of doing this. First we intersect the two RTEs above.

```
(def demo-seq [1 2 4/3 4 5 6.5 3 2])

(def rte-1 '(:and (:cat (:* Number) Double (:* Number))
                  (:cat (:* Number) Ratio (:* Number))))

(rte/match rte-1 demo-seq) ;; => true
```

We check whether it is a seqence of number which contains a `Double` and then a `Ratio`, or whether it contains a `Ratio` and then a `Double`. *I.e.*, the union of two RTEs.

```
(def demo-seq [1 2 4/3 4 5 6.5 3 2])

(def rte-2 '(:or (:cat (:* Number) Double (:* Number) Ratio (:* Number))
                 (:cat (:* Number) Ratio (:* Number) Double (:* Number))))

(rte/match rte-2 demo-seq) ;; => true
```

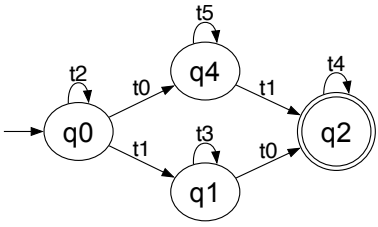
Visualizing RTEs

Are `rte-1` and `rte-2` really the same? Display the two DFAs corresponding to each.

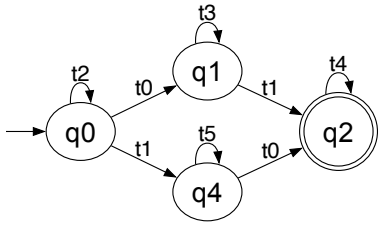
```
(def rte-1 '(:and (:cat (:* Number) Double (:* Number))
                  (:cat (:* Number) Ratio (:* Number))))

(dot/dfa-to-dot rte-1 :title "Dfa rte-1"
                :view true
                :state-legend false)

(def rte-2 '(:or (:cat (:* Number) Double (:* Number) Ratio (:* Number))
                 (:cat (:* Number) Ratio (:* Number) Double (:* Number))))

(dot/dfa-to-dot rte-2 :title "Dfa rte-2"
                :view true
                :state-legend false)
```

Visualizing RTEs

Are `rte-1` and `rte-2` really the same? Display the two DFAs corresponding to each.




```
t0= Double
t1= Ratio
t2= (and (not Double) (not Ratio) Number)
t3= (or (and (not Double) Number) Ratio)
t4= Number
t5= (or (and (not Ratio) Number) Double)
```

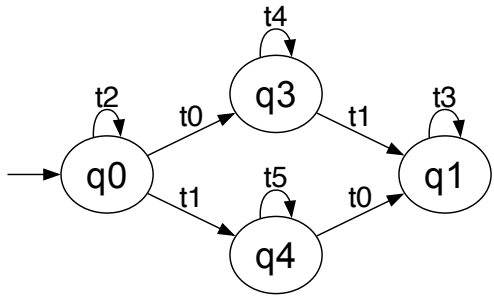



```
t0= Double
t1= Ratio
t2= (and (not Double) (not Ratio) Number)
t3= (or (and (not Ratio) Number) Double)
t4= Number
t5= (or (and (not Double) Number) Ratio)
```

The two DFAs are similar, but the type labels are slightly different. We compute the difference of the two DFAs by diplaying the XOR of the two RTEs.

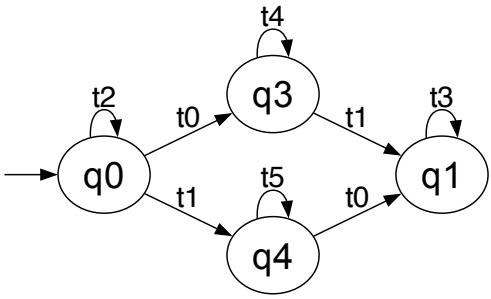
```
(dot/dfa-to-dot (rte/Xor rte-1 rte-2)
                :title "xor 1-2 dfa"
                :state-legend false
                :view true)
```

The two DFAs are similar, but the type labels are slightly different. We compute the difference of the two DFAs by diplaying the XOR of the two RTEs.




```
t0= Double
t1= Ratio
t2= (and (not Double) (not Ratio) Number)
t3= Number
t4= (or (and (not Ratio) Number) Double)
t5= (or (and (not Double) Number) Ratio)
```

We see that the DFA corresponding to the Xor of the two rtes has no accepting states, which means its language is empty. *I.e.*, there are no sequences which match one of the RTEs without also matching the other.




```
t0= Double
t1= Ratio
t2= (and (not Double) (not Ratio) Number)
t3= Number
t4= (or (and (not Ratio) Number) Double)
t5= (or (and (not Double) Number) Ratio)
```

```
(def rte-4 '(:and (:* Number)
                  (:cat (:* :sigma) Ratio (:* :sigma))
                  (:cat (:+ :sigma) (satisfies odd?) (:* :sigma))))

(xym/find-trace-map (rte/rte-to-dfa (rte/And-not rte-1 rte-4)))
;; => {true [:indeterminate [[:satisfiable Ratio]
;;                           [:indeterminate (and (not (satisfies odd?))
;;                                                Double)]]]}
```

Suppose we had made a mistake and declare an RTE which was not what we wanted. Suppose we define `rte-4` as below. We use the function `xym/find-trace-map` to describe a sequence which would match `rte-1` but not `rte-2`.

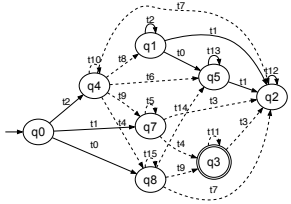
```
(def rte-4 '(:and (:* Number)
                  (:cat (:* :sigma) Ratio (:* :sigma))
                  (:cat (:+ :sigma) (satisfies odd?) (:* :sigma))))

(xym/find-trace-map (rte/rte-to-dfa (rte/And-not rte-1 rte-4)))
;; => {true [:indeterminate [[:satisfiable Ratio]
;;                           [:indeterminate (and (not (satisfies odd?))
;;                                                Double)]]]}
```

We see that `xym/find-trace-map` claims that a sequence consisting of `Ratio` followed by `(and (not (satisfies odd?)) Dobule)` would be such a sequence.

```
(def rte-4 '(:and (:* Number)
                  (:cat (:* :sigma) Ratio (:* :sigma))
                  (:cat (:+ :sigma) (satisfies odd?) (:* :sigma))))

(xym/find-trace-map (rte/rte-to-dfa (rte/And-not rte-1 rte-4)))
;; => {true [:indeterminate [[:satisfiable Ratio]
;;                           [:indeterminate (and (not (satisfies odd?))
;;                                                Double)]]]}
```

It specifies that the result is `:indeterminate` because not all the types in the sequences are known to be satisfiable. In particular `Ratio` is `:satisfiable`, but it it not known whether there is a value which is `Double` and fails to satisfy the predicate `odd?`. Of course we know that such a value exists, but it cannot be determined programmatically.

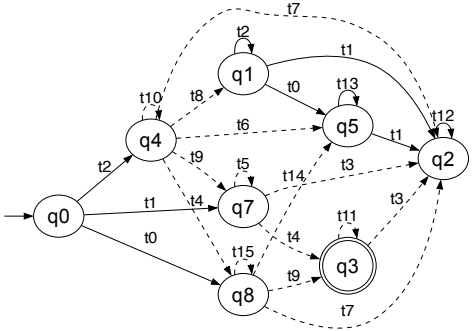
How does `xym/find-trace-map` work?




```
t0= Double
t1= Ratio
t2= (and (not Double) (not Ratio) Number)
t3= (or
      (and (not Double) (not Ratio) (satisfies odd?) Number)
      (and (satisfies odd?) Double)
      (and (satisfies odd?) Ratio))
t4= (and (not (satisfies odd?)) Double)
t5= (or
      (and (not (satisfies odd?)) (not Double) (not Ratio) Number)
      (and (not (satisfies odd?)) Ratio))
t6= (and (satisfies odd?) Double)
t7= (and (satisfies odd?) Ratio)
t8= (and (not Double) (not Ratio) (satisfies odd?) Number)
t9= (and (not (satisfies odd?)) Ratio)
t10= (and (not (satisfies odd?)) (not Double) (not Ratio) Number)
t11= (or
      (and (not (satisfies odd?)) (not Double) (not Ratio) Number)
      (and (not (satisfies odd?)) Double)
      (and (not (satisfies odd?)) Ratio))
t12= Number
t13= (or (and (not Ratio) Number) Double)
t14= (or (and (not Double) (not Ratio) (satisfies odd?) Number) (and (satisfies odd?) Double))
t15= (or
      (and (not (satisfies odd?)) (not Double) (not Ratio) Number)
      (and (not (satisfies odd?)) Double))
```

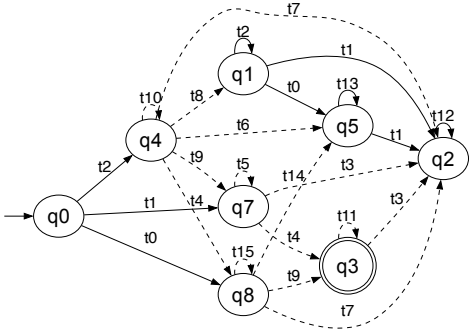
`xym/find-trace-map` examines the DFA generated by an RTE.

How does `xym/find-trace-map` work?



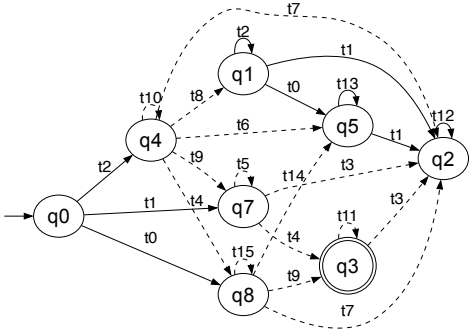
Then it asks whether there is a path from the initial state `q0` and some accepting state.

How does `xym/find-trace-map` work?



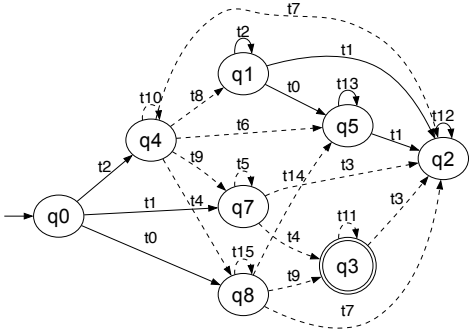
In particular is there a path which contains only satisfiable transitions, and if not then is there a path with some indeterminate transitions.

How does `xym/find-trace-map` work?



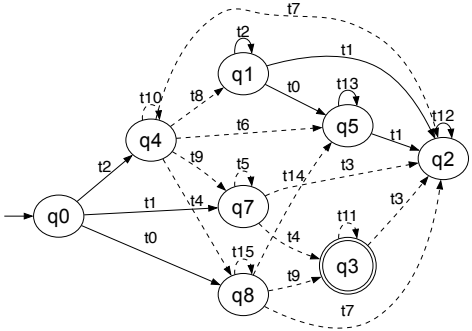
It answers this question by performing a Dijkstra search on the weighted graph of the DFA.

How does `xym/find-trace-map` work?



In the Dijkstra search, satisfiable transitions have a weight of 1, while indeterminate transitions have a weight equal to the number of states, 8 in the above examples.

How does `xym/find-trace-map` work?



Thus any path with a weight less than 8 must contain no indeterminate transitions, and any path with weight 8 or more must have at least one indeterminate transition.

Clojure Multiple-arity Functions

```
1 (defn f
2   ([a]   12)
3   ([a b]   13)
4   ([a b c]   14))
5
6 (f 0 0) ;; --> 13
```

The Clojure language supports functions of multiple arity. At the run-time selects which arity of the function to call.

```clojure
(defn f
  ([a]   12)
  ([a b]  13)
  ([a b [c d]]  14))

(f 0 [1 2]) ;; --> 14
```

Multiple-arity function definition also supports destructuring.

```
(defn f
   ([[a b] c d]  12)
   ([a [b c] d]  13)
   ([a b [c d]]  14))
```

However Clojure does not discriminate on the basis of structure, only on basis of arity. The following example contains two definitions for arity 3.

An attempt to define such a function will result in a compilation error.

```
;; Vain attempt to define destructuring function
(defn f
  ([[a b] c d]   12)
  ([a [b c] d]   13) => Can't have 2 overloads with same arity
  ([a b [c d]]   14))
```

```clojure
(defn f
  ([a]   12)
  ([^Boolean a b]  13)
  ([a b [c d]]  14))

(f true 2) ;; --> 13
(f 1 2)    ;; --> 13
```

So-called *type hints* may be specified but they have no semantic effect at run-time. The following type hint `Boolean` is ignored if the function is called with a first argument which is not of type `Boolean`.

```clojure
(defn f
  ([a]   12)
  ([^int? a b]  13)
  ([a b [c d]]  14))

(defn f
  ([a]   12)
  ([^Ratio a b]  13)
  ([a b [c d]]  14))
```

Some types are valid, some are not. E.g., `Ratio` is not recognized (rather you must use the verbose name `clojure.lang.Ratio`), and `int?` is a type `predicate` not type name.

The use of invalid type hints *usually* results in compilation errors.

```
(defn f
  ([a]    12)
  ([^Ratio a b]    13) => Unable to resolve classname: Ratio
  ([a b [c d]]    14))
```

```
(defn f
  ([a]    12)
  ([^int? a b]    13) => Unable to resolve classname: int?
  ([a b [c d]]    14))
```

However, some invalid type hints are silently ignored and seem to have no effect.

```
(defn f
  ([a]   12)
  ([a b]  13)
  ([a b [^int? c ^Ratio d]] 14))

(f 0 1 [3 52/17])     ;; --> 14
(f 0 1 ["hello" false]) ;; --> 14
```

Introducting `dsdefn` Destructuring Define Function

The simplest form of `dsdefn` allows the programmer to select the expressions to evaluate based on arity and structure.

```
(dsdefn f
  ([[a b] c d] 12)
  ([a [b c] d] 13)
  ([a b [c d]] 14))

(f [0 1] 2 3) ;; --> 12
(f 0 [1 2] 3) ;; --> 13
(f 0 1 [2 3]) ;; --> 14
```

If multiple desturing lambda lists are identical, a warning is issued. But when called, the function selects the first matching function to evaluate.

```
(dsdefn f
  ([[a b] c d] 12)
  ([a [b c] d] 13)
  ([a b [c d]] 14)
  ([a b [c d]] (println "fourth clause")
               15))

;; prints to *err*
;;   Unreachable code: ((println "fourth clause") 15)

(f 1 2 [10.0 20]) ;; --> 14
```

dsdefn also allows type hints with semantics. At function call time the first matter which matches the types of the given argument values is selected and evaluated.

```
(dsdefn f
  ([[a b] c d] 12)
  ([a [b c] d] 13)
  ([a b [^Ratio c d]] 14)
  ([a b [^Integer c d]] 15)
  ([a b [^Number c d]] 16)
  ([a b [^Double c d]] 17))


(f 1 2 [10.0 20]) ;; --> 16
(f 1 2 [2/3 3]) ;; --> 14

(f 1 2 [10 20]) ;; --> 16 !! ATTENTION !!

(instance? Integer 10) ;; --> false
(instance? Long 10)  ;; --> true
(instance? Number 10) ;; --> true
```

Attention the value `10` is not of type `Integer` in Clojure. Rather to match an integer we must use a type predicate: `int?` (to exclude big-nums) or perhaps `integer?` (to include big-nums).

```
;; Vain attempt
(dsdefn f
  ([[a b] c d] 12)
  ([a [b c] d] 13)
  ([a b [^Ratio c d]] 14)
  ([a b [^int? c d]] 15)
  ([a b [^Number c d]] 16)
  ([a b [^Double c d]] 17))
```

Attention the value `10` is not of type `Integer` in Clojure. Rather to match an integer we must use a type predicate: `int?` (to exclude big-nums) or perhaps `integer?` (to include big-nums).

```
;; Vain attempt
(dsdefn f
  ([[a b] c d] 12)
  ([a [b c] d] 13)
  ([a b [^Ratio c d]] 14)
  ([a b [^int? c d]] 15) => clojure.lang.int?
  ([a b [^Number c d]] 16)
  ([a b [^Double c d]] 17))
```

Rather than a raw type predicate we have to use an alternate syntax. `(satisfies int?)`. Unfortunately, we cannot use `(satisfies int?)` directly in the Clojure type-hint syntax. Clojure does not allow us to use a non-symbol a as type-hint.

```
;; Vain attempt
(dsdefn f
  ([[a b] c d] 12)
  ([a [b c] d] 13)
  ([a b [^Ratio c d]] 14)
  ([a b [^(satisfies int?) c d]] 15)
  ([a b [^Number c d]] 16)
  ([a b [^Double c d]] 17))
```

Rather than a raw type predicate we have to use an alternate syntax. `(satisfies int?)`. Unfortunately, we cannot use `(satisfies int?)` directly in the Clojure type-hint syntax. Clojure does not allow us to use a non-symbol a as type-hint.

```
(dsdefn f
  ([[a b] c d] 12)
  ([a [b c] d] 13)
  ([a b [^Ratio c d]] 14)
  ([a b [^(satisfies int?) c d]] 15)
 => java.lang.IllegalArgumentException: Metadata must be Symbol,Keyword,String,Vector or Map
Metadata must be Symbol,Keyword,String,Vector or Map
  ([a b [^Number c d]] 16)
  ([a b [^Double c d]] 17))
```

Because of this limitation, we provide an alternative/supplementary syntax. We may preceed the argument list vector with a hash as meta data. The hash associates variable names declared in the argument vector with type declarations.

```
(dsdefn f
  ([[a b] c d] 12)
  ([a [b c] d] 13)
  ([a b [^Ratio c d]] 14)
  (^{c (satisfies int?)} [a b [c d]] 15)
  ([a b [^Double c d]] 16))

(f 1 2 [10/3 20]) ;; --> 14
(f 1 2 [10 20]) ;; --> 15
(f 1 2 [10.0 20]) ;; --> 16
(f 1 2 [false 20]) ;; ERROR: No pattern matching given sequence
                           {:sequence (1 2 [false 20])}
```

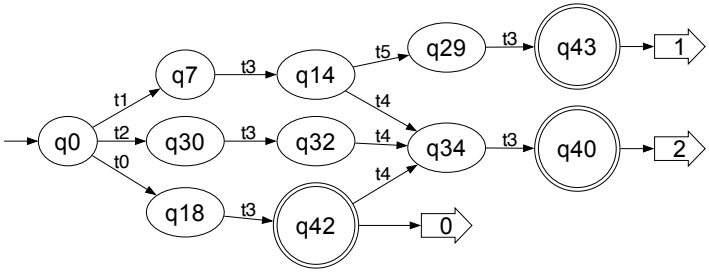
Because of this limitation, we provide an alternative/supplementary syntax. We may preceed the argument list vector with a hash as meta data. The hash associates variable names declared in the argument vector with type declarations.

```
1. Unhandled clojure.lang.ExceptionInfo
   No pattern matching given sequence
   {:sequence (1 2 [false 20])}
                      REPL:  109  reclojure2025/fn/fn
               RestFn.java:  439  clojure.lang.RestFn/invoke
                      REPL:  119  reclojure2025/eval65437
                      REPL:  119  reclojure2025/eval65437
             Compiler.java: 7700  clojure.lang.Compiler/eval
    interruptible_eval.clj:  106  nrepl.middleware.interruptible-eval/evalu
    interruptible_eval.clj:  101  nrepl.middleware.interruptible-eval/evalu
               session.clj:  230  nrepl.middleware.session/session-exec/ses
       SessionThread.java:   21  nrepl.SessionThread/run
```

In selecting which clause to evaluate, a computation is done using a DFA. The DFA is build by the macro expansion (when the macro expansion is evaluated). When the function `f` is called the DFA is traversed once to determine which piece of code to evaluate.

```
1 (dsdefn f
2   ([^Number a b] 14)
3   (^{a Boolean
4      c (satisfies int?)} [a b c d] 15)
5   ( [a b ^Double c d] 16))
```

In selecting which clause to evaluate, a computation is done using a DFA. The DFA is build by the macro expansion (when the macro expansion is evaluated). When the function `f` is called the DFA is traversed once to determine which piece of code to evaluate.




```
t0= Number
t1= Boolean
t2= (and (not Boolean) (not Number))
t3= :sigma
t4= Double
t5= (or Byte Integer Long Short)
```

The `rte-case` Macro