core.logic

A Tutorial Reconstruction

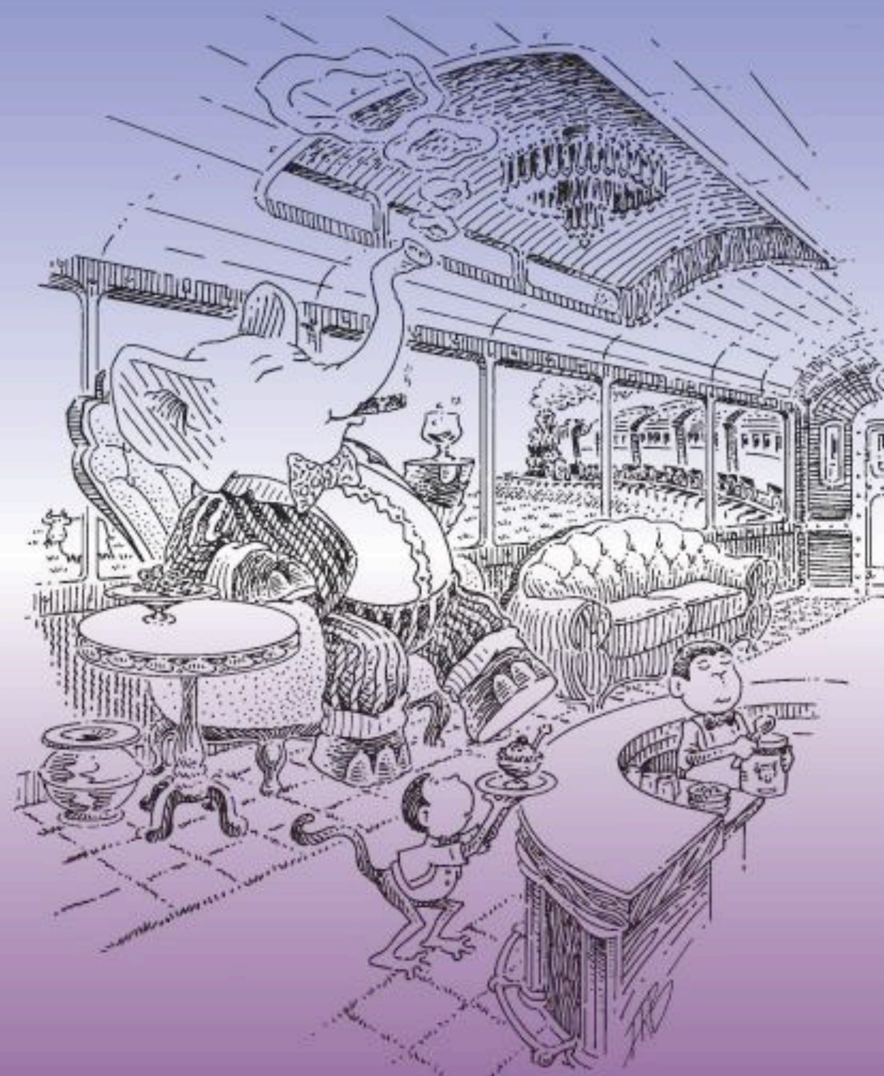
Warren's Abstract Machine

A Tutorial Reconstruction

Hassan Aït-Kaci

An MIT Press Classic

The Reasoned Schemer



Daniel P. Friedman, William E. Byrd,
and Oleg Kiselyov

Kanren

miniKanren

~200 LOC

```
(run* [q]
  (== q true))

→ (true)
```

We’re not in Kansas anymore

```
(run* [q]
  (== q true))

→ (true)
```

logic variable

```
(run* [q]
  (== q true))

→ (true)
```

goal /
relation

```
(run* [q]
  (== q true))

→ (true)
```

```
(defn == [term1 term2]
  (fn [a]
    ...))
```

```
(defn == [term1 term2]
   (fn [a]
     ...))
```

```
(run* [q]
  (== q true))

→ (true)
```

results

```
(run* [q]
  (== q true))
```


```
→ (true)
```

```
(run* [q]
  (== q true))

→ (true)
```

flipped variables to unify goal ... order doesn't matter!

```
(run* [q]
  (== true q))

→ (true)
```

```
(run* [q]
  (== true q)
  (== q false))

→ ()
```

flip the order of the goals

```
(run* [q]
  (== q false)
  (== true q))

→ ()
```

```
(run* [q]
  (== q true)
  (== q true))

→ (true)
```

single assignment

```
(run* [q]
  (fresh [x y]
    (== [1 y] [x 2])
    (== q [x y])))

→ ([1 2])
```

```
(run* [q]
  (fresh [x y]
    (== [1 y] [x 2])
    (== q [x y])))

→ ([1 2])
```

fresh

fresh

- Don’t be scared

fresh

- Don’t be scared
- Like Scheme let – delimits a lexical scope

fresh

- Don’t be scared
- Like Scheme let – delimits a lexical scope
- Introduces logic variables

fresh

- Don’t be scared
- Like Scheme let – delimits a lexical scope
- Introduces logic variables
- Unlike Scheme let you don’t use it to actually bind these locals to values

```
(run* [q]
  (fresh [x y]
    (== [1 y] [x 2])
    (== q [x y])))

→ ([1 2])
```

Unification

Unification

- Attempt to find the solution that will make two terms equal

Unification

- Attempt to find the solution that will make two terms equal
  - trivial cases: `(== 1 1)`, `(== 1 2)`

Unification

- Attempt to find the solution that will make two terms equal
  - trivial cases: `(== 1 1)`, `(== 1 2)`
  - but also complex terms as we saw on previous on slide

terms are sexprs

Unification

Unification

- A map-like structure holds bindings between logic variables and their values

Unification

- A map-like structure holds bindings between logic variables and their values
- We want to minimize book-keeping when backtracking - so we need some kind of persistent data structure ...

Unification

- A map-like structure holds bindings between logic variables and their values
- We want to minimize book-keeping when backtracking - so we need some kind of persistent data structure ...
  - a list!

Triangular Substitution

Triangular Substitution

- Substitution map s is a list of two element tuples

Triangular Substitution

- Substitution map s is a list of two element tuples
- In each tuple, variables always appear on the left-hand side

Triangular Substitution

- Substitution map s is a list of two element tuples
- In each tuple, variables always appear on the left-hand side
- Lookup is a linear traversal of the list that examines the left-hand side for a match

```
s = ((z . true) (y . z) (x . y))
```

```
(run* [q]
  (== q true))

→ (true)
```

```
s = ((q . true))
```

```
(run* [q]
  (fresh [x y]
    (== x y)
    (== y q)
    (== x true)))

→ (true)
```

```
(run* [q]
  (fresh [x y]
    (== x y)
    (== y q)
    (== x true)))

→ (true)
```

```
s = ((x . y))
```

```
(run* [q]
  (fresh [x y]
    (== x y)
    (== y q)
    (== x true)))

→ (true)
```

```
s = ((y . q)(x . y))
```

```
(run* [q]
  (fresh [x y]
    (== x y)
    (== y q)
    (== x true)))

→ (true)
```

```
s = ((q . true)(y .
q)
     (x . y))
```

```
(== x 1)
```

```
s = ((y . q) (x . y))
```

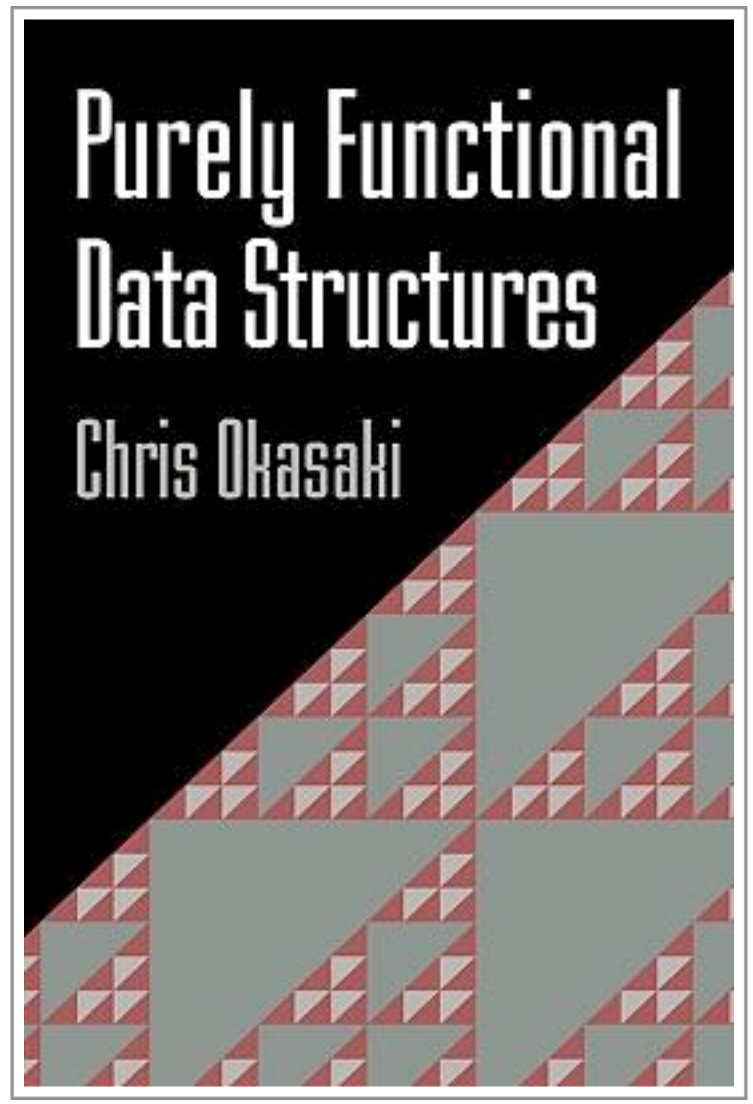
$$\frac{\texttt{(== x 1)}}{\texttt{s = ((y . q) (x . y))}}$$

$$\frac{\texttt{(== y 1)}}{\texttt{s = ((y . q) (x . y))}}$$

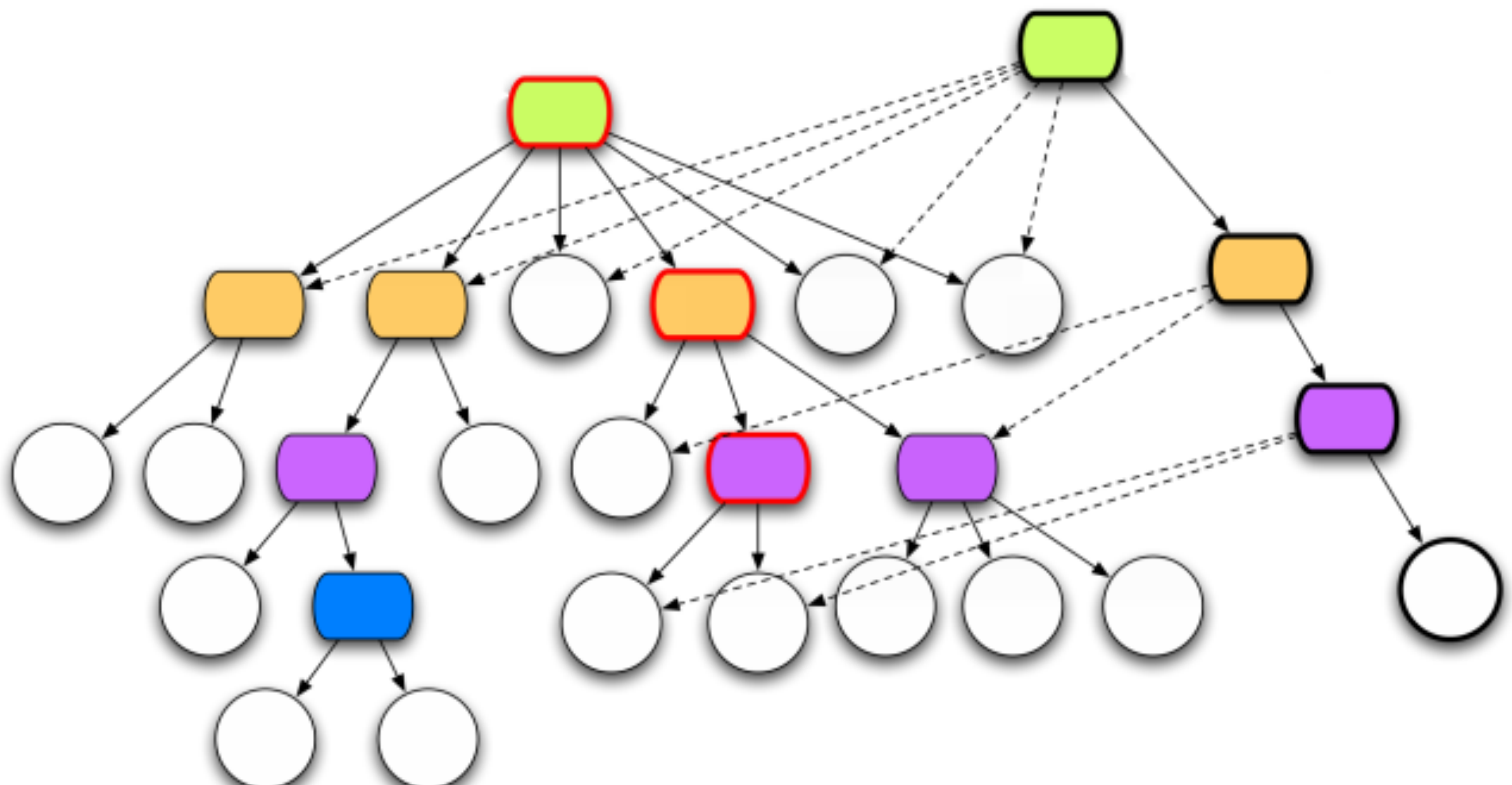
$$\frac{\texttt{(== q 1)}}{\texttt{s = ((y . q) (x . y))}}$$

walk

```
(deftype Substitution [s ...] ...)
```



Purely Functional
Data Structures
Chris Okasaki



PersistentHashMap

```
(deftype Substitution [s ...] ...)
```

```
(walk [this v]
  (loop [lv v [v vp] (find s v)]
    (cond
      (nil? v) lv
      (not (lvar? vp)) vp
      :else (recur vp (find s vp)))))
```

Substitutions

```
(walk [this v]
  (loop [lv v [v vp] (find s v)]
    (cond
      (nil? v) lv
      (not (lvar? vp)) vp
      :else (recur vp (find s vp)))))
```

```
(walk [this v]
  (loop [lv v [v vp] (find s v)]
    (cond
      (nil? v) lv
      (not (lvar? vp)) vp
      :else (recur vp (find s vp)))))
```

```clojure
(walk [this v]
  (loop [lv v [v vp :as me] (find s v)]
    (cond
      (nil? me) lv
      (not (lvar? vp)) vp
      :else (recur vp (find s vp)))))
```

```
(walk [this v]
  (loop [lv v [v vp :as me] (find s v)]
    (cond
      (nil? me) lv
      (not (lvar? vp)) vp
      :else (recur vp (find s vp)))))
```

```
(walk [this v]
  (loop [lv v [v vp :as me] (find s v)]
    (cond
      (nil? me) lv
      (not (lvar? vp)) vp
      :else (recur vp (find s vp)))))
```

unify

Substitutions

```
(defn unify [s u v]
  (if (identical? u v)
    s
    (let [u (walk s u)
          v (walk s v)]
      (if (identical? u v)
        s
        (unify-terms u v s)))))
```

terms

```
(defn unify [s u v]
  (if (identical? u v)
    s
    (let [u (walk s u)
          v (walk s v)]
      (if (identical? u v)
        s
        (unify-terms u v s)))))
```

```
(defn unify [s u v]
  (if (identical? u v)
    s
    (let [u (walk s u)
          v (walk s v)]
      (if (identical? u v)
        s
        (unify-terms u v s)))))
```

```
(defn unify [s u v]
  (if (identical? u v)
    s
    (let [u (walk s u)
          v (walk s v)]
      (if (identical? u v)
        s
        (unify-terms u v s)))))
```

```
(defn unify [s u v]
  (if (identical? u v)
    s
    (let [u (walk s u)
          v (walk s v)]
      (if (identical? u v)
        s
        (unify-terms u v s)))))
```

```clojure
(defn unify [s u v]
  (if (identical? u v)
    s
    (let [u (walk s u)
          v (walk s v)]
      (if (identical? u v)
        s
        (unify-terms u v s)))))
```

core.logic vs. mK unify

core.logic vs. mK unify

- miniKanren unification handle Scheme types - list is the only type that triggers recursion during unification

core.logic vs. mK unify

- miniKanren unification handle Scheme types - list is the only type that triggers recursion during unification
- Clojure has many more datatypes - maps, sets, vectors

core.logic vs. mK unify

- miniKanren unification handle Scheme types - list is the only type that triggers recursion during unification
- Clojure has many more datatypes - maps, sets, vectors
- Unification needs to be polymorphic!

core.logic vs. mK unify

- miniKanren unification handle Scheme types - list is the only type that triggers recursion during unification
- Clojure has many more datatypes - maps, sets, vectors
- Unification needs to be polymorphic!
- and open!

```scheme
(define unify
  (lambda (u v s)
    (let ((u (walk u s))
          (v (walk v s)))
      (cond
        ((eq? u v) s)
        ((var? u) (ext-s-check u v s))
        ((var? v) (ext-s-check v u s))
        ((and (pair? u) (pair? v))
         (let ((s (unify (car u) (car v) s)))
           (and s (unify (cdr u) (cdr v) s))))
        ((equal? u v) s)
        (else #f)))))
```

```
(run* [q]
  (== q true))

→ (true)
```

```
(run* [q]
  (fn [a]
    (unify a q true)))

→ (true)
```

Substitutions

```
(run* [q]
  (fn [a]
    (unify a q true)))

→ (true)
```

```
(run* [q]    u   v
  (fn [a]
    (unify a q true)))

→ (true)
```

```
(defn unify [s u v]
  (if (identical? u v)
    s
    (let [u (walk s u)
          v (walk s v)]
      (if (identical? u v)
        s
        (unify-terms u v s))))))
```

```
(defprotocol IUnifyTerms
  (unify-terms [u v s]))
```

```
(defprotocol IUnifyTerms
  (unify-terms [u v s]))
```

terms

```
(defprotocol IUnifyTerms
  (unify-terms [u v s]))
```

Substitutions

Protocols

Protocols

- Create polymorphic functions that dispatch on type of first argument

Protocols

- Create polymorphic functions that dispatch on type of first argument
- You can extend types to a protocol even if it’s original definition did not implement it

Protocols

- Create polymorphic functions that dispatch on type of first argument
- You can extend types to a protocol even if it’s original definition did not implement it
- Special handling of nil

Protocols

- Create polymorphic functions that dispatch on type of first argument
- You can extend types to a protocol even if it’s original definition did not implement it
- Special handling of nil
- Can define a default implementation by extending a protocol to Object

```
(deftype LVar [name ...]
  ...
  IUnifyTerms
  (unify-terms [u v s]
    (unify-with-lvar v u s))
  ...
  )
```

```
(deftype LVar [name ...]
  ...
  IUnifyTerms
  (unify-terms [u v s]
    (unify-with-lvar v u s))
  ...
  )
```

flip!

```
(extend-protocol IUnifyWithLVar
  nil
  (unify-with-lvar [v u s]
    (ext-no-check s u v))

  Object
  (unify-with-lvar [v u s]
    (ext s u v)))
```

```
(extend-protocol IUnifyWithLVar
  nil
  (unify-with-lvar [v u s]
    (ext-no-check s u v))

  Object
  (unify-with-lvar [v u s]
    (ext s u v)))
```

```
(run* [q]
  (== 1 1))

→ (_.0)
```

```clojure
(extend-protocol IUnifyTerms
  nil
  (unify-terms [u v s]
    (unify-with-nil v u s))

  Object
  (unify-terms [u v s]
    (unify-with-object v u s))

  clojure.lang.Sequential
  (unify-terms [u v s]
    (unify-with-seq v u s))

  clojure.lang.IPersistentMap
  (unify-terms [u v s]
    (unify-with-map v u s)))
```

```
(extend-protocol IUnifyTerms
  nil
  (unify-terms [u v s]
    (unify-with-nil v u s))

  Object
  (unify-terms [u v s]
    (unify-with-object v u s))

  clojure.lang.Sequential
  (unify-terms [u v s]
    (unify-with-seq v u s))

  clojure.lang.IPersistentMap
  (unify-terms [u v s]
    (unify-with-map v u s)))
```

```clojure
(extend-protocol IUnifyWithObject
  nil
  (unify-with-object [v u s] nil)

  Object
  (unify-with-object [v u s]
    (if (= u v) s nil)))
```

```
(extend-protocol IUnifyWithObject
  nil
  (unify-with-object [v u s] nil)

  Object
  (unify-with-object [v u s]
    (if (= u v) s nil)))
```

```
(extend-protocol IUnifyWithObject
  nil
  (unify-with-object [v u s] nil)

  Object
  (unify-with-object [v u s]
    (if (= u v) s nil)))
```

FAIL!

```clojure
(extend-protocol IUnifyTerms
  nil
  (unify-terms [u v s]
    (unify-with-nil v u s))

  Object
  (unify-terms [u v s]
    (unify-with-object v u s))

  clojure.lang.Sequential
  (unify-terms [u v s]
    (unify-with-seq v u s))

  clojure.lang.IPersistentMap
  (unify-terms [u v s]
    (unify-with-map v u s))))
```

```clojure
(extend-protocol IUnifyTerms
  nil
  (unify-terms [u v s]
    (unify-with-nil v u s))

  Object
  (unify-terms [u v s]
    (unify-with-object v u s))

  clojure.lang.Sequential
  (unify-terms [u v s]
    (unify-with-seq v u s))

  clojure.lang.IPersistentMap
  (unify-terms [u v s]
    (unify-with-map v u s))))
```

recursive cases

```clojure
(extend-protocol IUnifyWithSequential
  nil
  (unify-with-seq [v u s] nil)

  Object
  (unify-with-seq [v u s] nil)

  clojure.lang.Sequential
  (unify-with-seq [v u s]
    (loop [u u v v s s]
      (if (seq u)
        (if (seq v)
          (if-let [s (unify s (first u) (first v))]
            (recur (next u) (next v) s)
            nil)
          nil)
        (if (seq v) nil s)))))
```

```clojure
(extend-protocol IUnifyWithSequential
  nil
  (unify-with-seq [v u s] nil)

  Object
  (unify-with-seq [v u s] nil)

  clojure.lang.Sequential
  (unify-with-seq [v u s]
    (loop [u u v v s s]
      (if (seq u)
        (if (seq v)
          (if-let [s (unify s (first u) (first v))]
            (recur (next u) (next v) s)
            nil)
          nil)
        (if (seq v) nil s)))))
```

reification

```clojure
(run* [q]
  (fresh [x y]
    (== x 1)
    (== y 2)
    (== q [x y])))

→ ([1 2])
```

```
(run* [q]
  (fresh [x y]
    (== x 1)
    (== y 2)
    (== q [x y])))

→ ([<lvar x> <lvar y>])
```

```
s = ((q . [x y]) (y . 2) (x .
1))
```

walk is not recursive on the term!

walk*

```
(defn walk* [s v]
  (let [v (walk s v)]
    (walk-term v s)))
```

```
(defprotocol IWalkTerm
  (walk-term [v s]))
```

```
(run* [q]
  (== true true))

→ (_.0)
```

```clojure
(defn reify-lvar-name [s]
  (symbol (str "_." (count s))))

(defn -reify* [s v]
  (let [v (walk s v)]
    (reify-term v s)))

(defn -reify [s v]
  (let [v (walk* s v)]
    (walk* (-reify* empty-s v) v)))
```

```
(defn reify-lvar-name [s]
  (symbol (str "_." (count s))))

(defn -reify* [s v]
  (let [v (walk s v)]
    (reify-term v s)))

(defn -reify [s v]
  (let [v (walk* s v)]
    (walk* (-reify* empty-s v) v)))
```

```
(defn reify-lvar-name [s]
  (symbol (str ”_.” (count s))))

(defn -reify* [s v]
  (let [v (walk s v)]
    (reify-term v s)))

(defn -reify [s v]
  (let [v (walk* s v)]
    (walk* (-reify* empty-s v) v)))
```

```
(defn reify-lvar-name [s]
  (symbol (str "_." (count s))))

(defn -reify* [s v]
  (let [v (walk s v)]
    (reify-term v s)))

(defn -reify [s v]
  (let [v (walk* s v)]
    (walk* (-reify* empty-s v) v)))
```

```
(defprotocol IReifyTerm
  (reify-term [v s]))
```

```
(deftype LVar [name ...]
  ...
  IReifyTerm
  (reify-term [v s]
    (if *reify-vars*
      (ext s v (reify-lvar-name s))
      (ext s v (:oname v))))
  ...)
```

∞

a-inf

a-inf

- nil (mzero)

a-inf

- nil (mzero)
- a substitution (unit)

a-inf

- nil (mzero)
- a substitution (unit)
- a choice (a stream)

a-inf

- nil (mzero)
- a substitution (unit)
- a choice (a stream)
- a thunk

bind

```
(run* [q]
  (== q true)
  (== q true))
```

```
(run* [q]
  (fn [a] (unify ...))
  (fn [a] (unify ...)))
```

```
(run* [q]
  (bind
    (bind empty-s
      (fn [a] (unify ...)))
    (fn [a] (unify ...))))
```

```
(run* [q]
  (bind
    ((q . true))
    (fn [a] (unify ...))))
```

```
(run* [q]
   ((q . true)))
```

```
(run* [q]
  (== q true)
  (== q false))
```

```
(run* [q]
  (bind
    ((q . true))
    (fn [a] (unify ...))))
```

```
(run* [q]
  nil)
```

```
(defprotocol IBind
  (bind [a-inf g]))
```

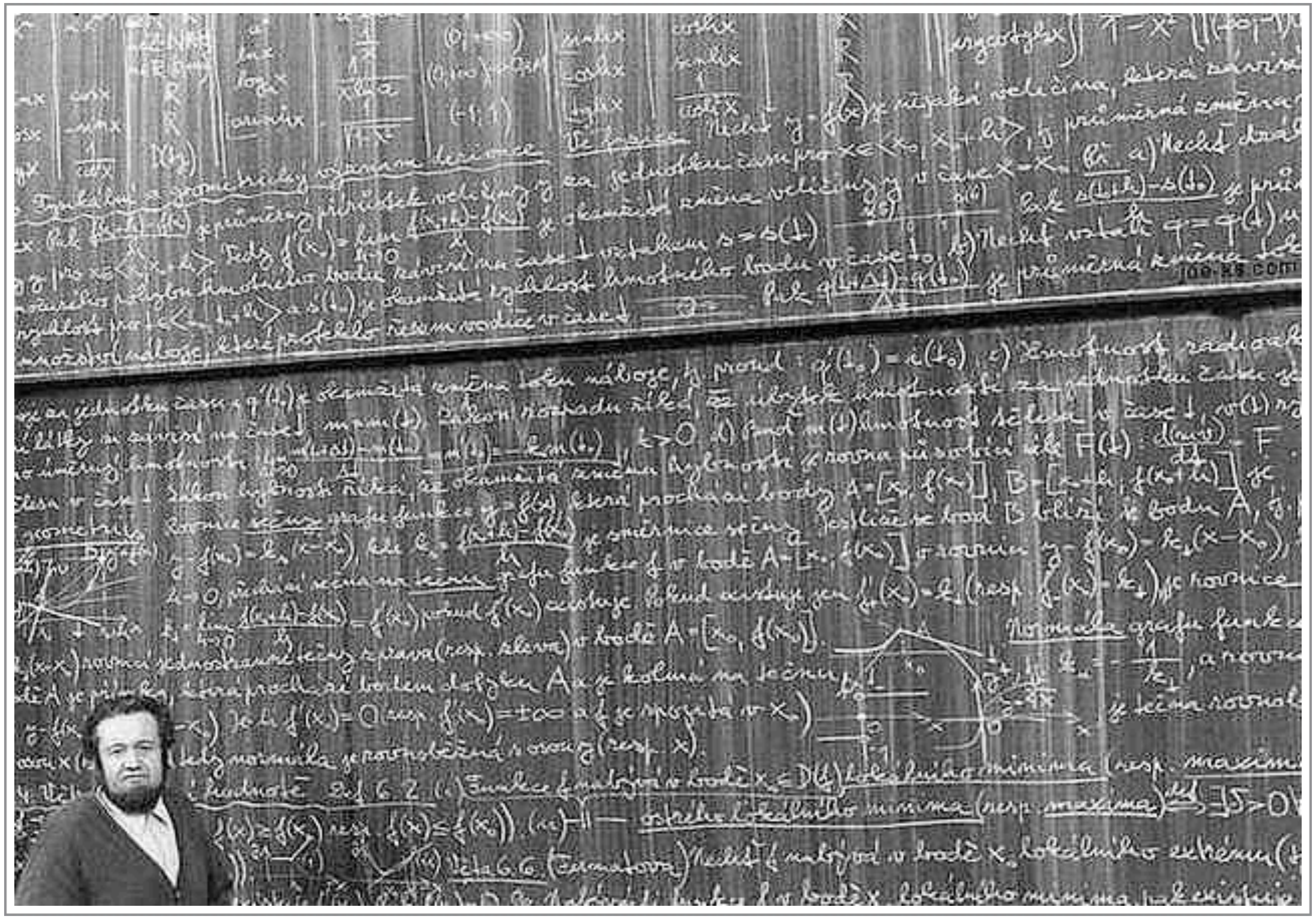
```
(deftype Substitutions [s ...]
  ...
  IBind
  (bind [this g]
    (g this))
  ...)
```

```
(extend-protocol IBind
  nil
  (bind [_ g] nil))
```

```
(run* [q]
  (fresh [x y]
    g0
    g1))
```

```
(run* [q]
  (let [x (lvar ‘x)
        y (lvar ‘y)]
    (bind
      (bind empty-s g0)
        g1))))
```

```
(run* [q]
  (let [x (lvar ‘x)
        y (lvar ‘y)]
    (bind
      (bind
        (bind empty-s g0)
        g1))
      (fn [a]
        (-reify a q))))
```



```
(run* [q]
  (conde
    [(== q 'tea)]
    [(== q 'coffee)]))

→ (tea coffee)
```

```
(run* [q]
  (bind empty-s
    (fn [a]
      (mplus
        (bind a (fn [a] (unify q ‘tea)))
        (fn []
          (bind a (fn [a] (unify q ‘coffee)))))))))
```

```
(defprotocol IMPlus
  (mplus [a-inf f]))
```

```
(defprotocol IMPlus
  (mplus [a-inf f]))
```

thunk

```
(deftype Substitutions [s ...]
  ...
  IMPlus
  (mplus [this f]
    (choice this f))
  ...)
```

```
(run* [q]
  (conde
    [(== q ‘tea)]
    [(== q ‘coffee)])
  (== q ‘tea))

→ (tea)
```

produces a **stream**!

```
(run* [q]
  (conde
    [(== q 'tea)]
    [(== q 'coffee)])
  (== q 'tea))

→ (tea)
```

```
(choice a-inf f)
```

rest of the stream in thunk

```
(choice a-inf f)
```

could be a stream too!

```
(choice a-inf f)
```

```
(deftype Choice [a-inf f]
  ....
  IBind
  (bind [this g]
    (mplus (g a-inf) (-inc (bind f g))))
  ...)
```

```
(deftype Choice [a-inf f]
  ....
  IBind
  (bind [this g]
    (mplus (g a-inf) (-inc (bind f g))))
  ...)
```

seems familiar somehow ...


icanhasGIF.com

map!

wait!

```
(deftype Choice [a-inf f]
  ....
  IMPlus
  (mplus [this fp]
    (Choice. a-inf (fn [] (mplus (fp) f))))
  ...)
```


icanhasGIF.com

mapcat!

```
(deftype Choice [a-inf f]
  ....
  IMPlus
  (mplus [this fp]
    (Choice. a (fn [] (mplus (fp) f))))
  ...)
```

WTF!

```
(deftype Choice [a-inf f]
  ....
  IMPlus
  (mplus [this fp]
    (Choice. a (fn [] (mplus (fp) f))))
  ...)
```

```
(run 1 [q]
  (conde
    [(nevero)]
    [(== q ‘tea)])
  (== q ‘tea))

→ (tea)
```

```
(extend-type clojure.lang.Fn
  IBind
  (bind [this g]
    (-inc (bind (this) g)))
  IMPlus
  (mplus [this f]
    (-inc (mplus (f) this)))
  ...)
```

–inc

-inc

- fresh & conde are wrapped in a -inc

-inc

- fresh & conde are wrapped in a -inc
- all goals are built upon fresh & conde

-inc

- fresh & conde are wrapped in a -inc
- all goals are built upon fresh & conde
- the -incs prevents bad goals from stealing time from good ones that have results

-inc

- fresh & conde are wrapped in a -inc
- all goals are built upon fresh & conde
- the -incs prevents bad goals from stealing time from good ones that have results
- a formal proof of why we need them in the previous slide would be nice ;)

run

run

- calls take* on the stream (a-inf) produced by running the goals

run

- calls take* on the stream (a-inf) produced by running the goals
- a trampoline that extracts reified values out of the stream (a-inf)!

run

- calls take* on the stream (a-inf) produced by running the goals
- a trampoline that extracts reified values out of the stream (a-inf)!
- “distributed” trampoline, jumps between different branches!

```
(defmacro run*
  [& goals]
  `(run false ~@goals))
```

```
(defmacro run
  [n & goals]
  `(doall (solve ~n ~@goals)))
```

```
(defmacro solve [& [n [x :as bindings] & goals]]
  (if (> (count bindings) 1)
    `(solve ~n [q#] (fresh ~bindings ~@goals (== q# ~bindings)))
    `(let [xs# (take* (fn []
                        ((fresh [~x]
                           ~@goals
                           (reifyg ~x))
                         empty-s)))]
       (if ~n
         (take ~n xs#)
         xs#))))
```

- no promises about order

- no promises about order
- miniKanren is a fundamentally concurrent design

- no promises about order
- miniKanren is a fundamentally concurrent design
- let’s exploit it!

Thank you!

Questions?