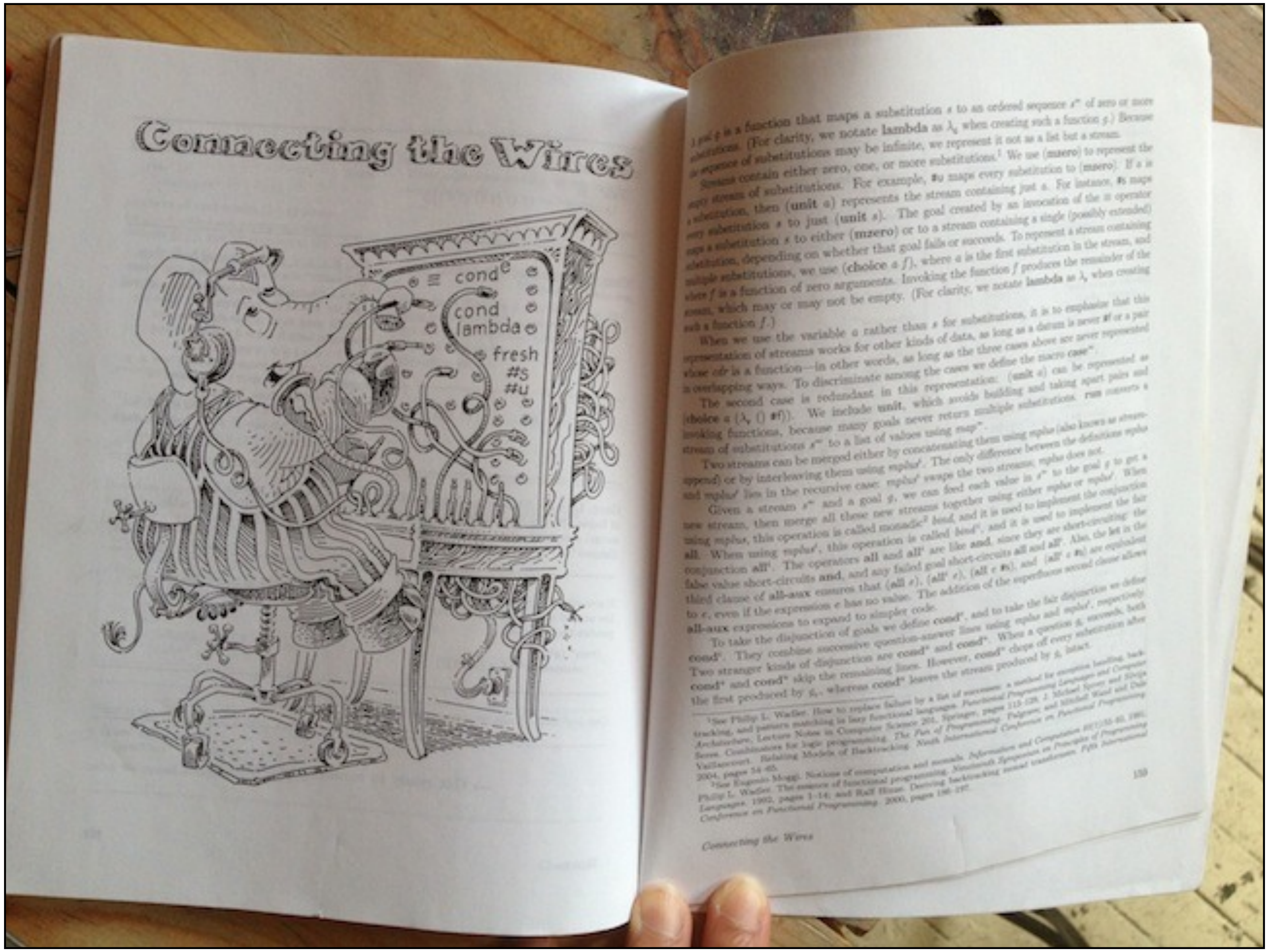
The Architecture of core.logic

Clojure/West 2013

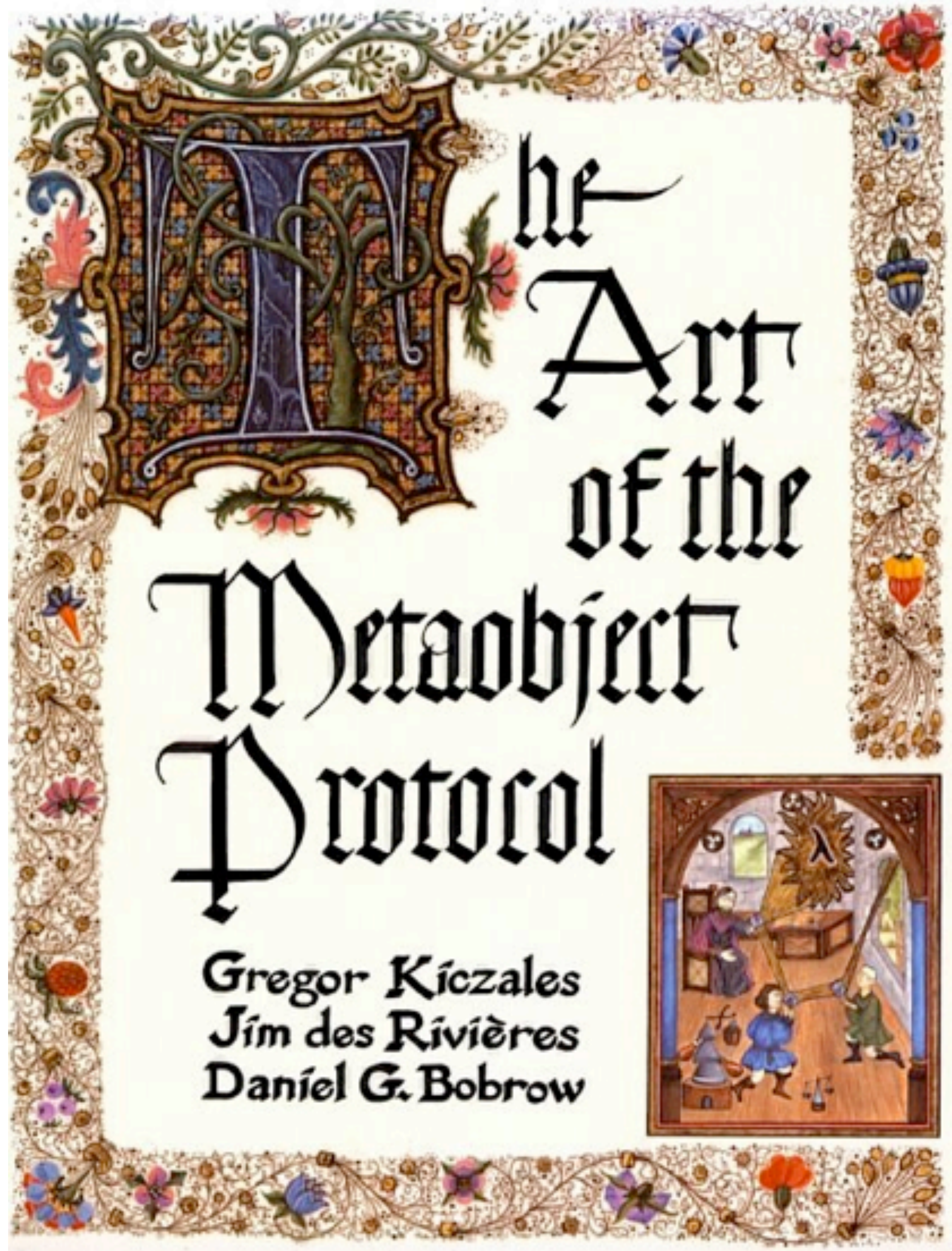

Connecting the Wires
cond
cond
lambda
fresh
#s
#u
Connecting the Wires

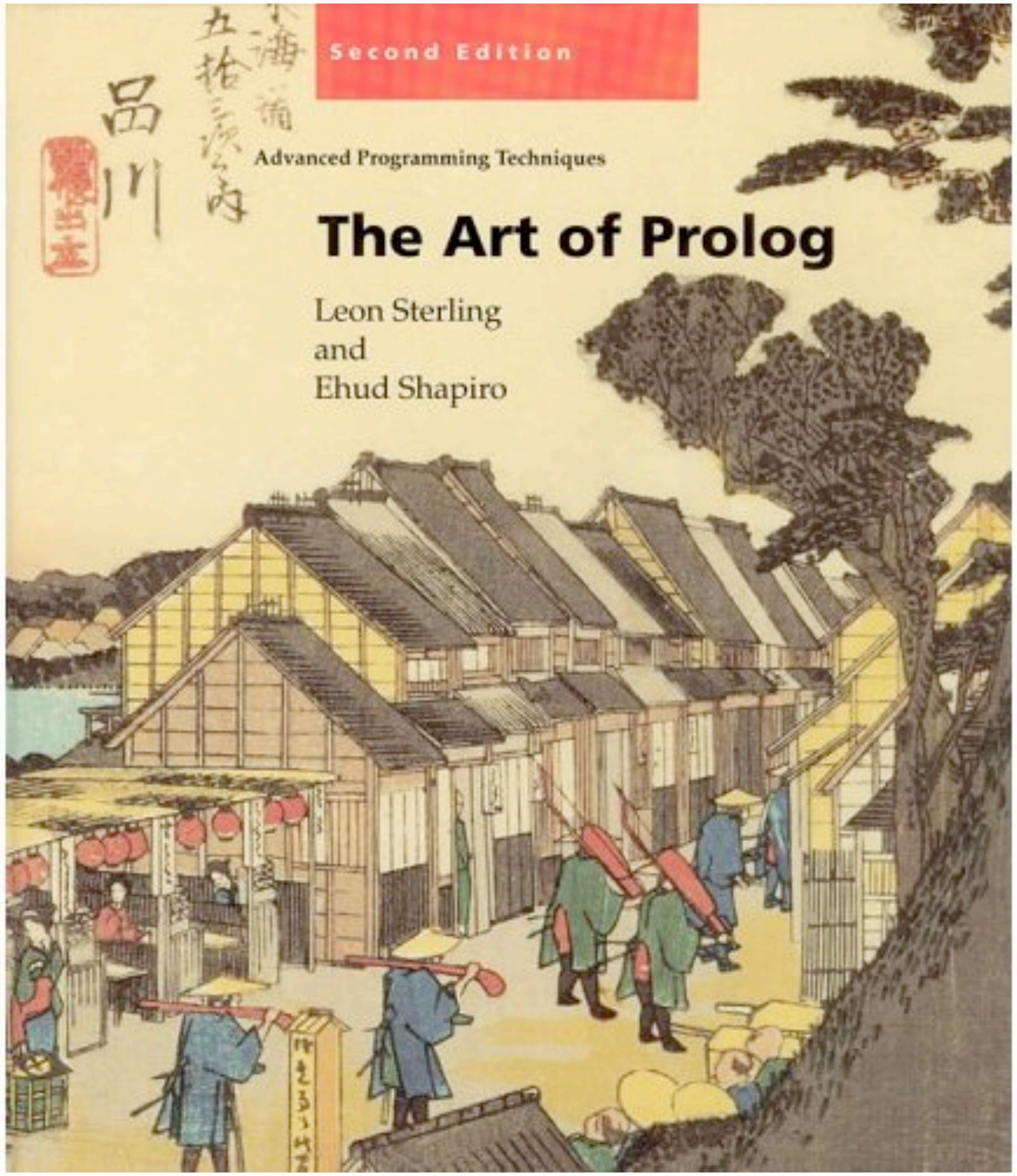
Scheme

```scheme
(define-syntax case-inf
  (syntax-rules ()
    ((_ e (() e0) ((f^) e1) ((c^) e2) ((c f) e3))
     (let ((c-inf e))
       (cond
         ((not c-inf) e0)
         ((procedure? c-inf)  (let ((f^ c-inf)) e1))
         ((not (and (pair? c-inf)
                (procedure? (cdr c-inf))))
          (let ((c^ c-inf)) e2))
         (else (let ((c (car c-inf)) (f (cdr c-inf)))
                e3)))))))

(define bind
  (lambda (c-inf g)
    (case-inf c-inf
      (() (mzero))
      ((f) (inc (bind (f) g)))
      ((c) (g c))
      ((c f) (mplus (g c) (lambdaf@ () (bind (f) g)))))))

(define mplus
  (lambda (c-inf f)
    (case-inf c-inf
      (() (f))
      ((f^) (inc (mplus (f) f^)))
      ((c) (choice c f))
      ((c f^) (choice c (lambdaf@ () (mplus (f) f^)))))))

(define take
  (lambda (n f)
    (cond
      ((and n (zero? n)) '())
      (else
       (case-inf (f)
         (() '())
         ((f) (take n f))
         ((c) (cons c '()))
         ((c f) (cons c (take (and n (- n 1)) f))))))))
```

Clojure

```clojure
(deftype Substitutions [...]
  ...
  IBind
  (bind [this g]
    (g this))
  IMPlus
  (mplus [this f]
    (choice this f))
  ITake
  (take* [this] this))

(deftype Choice [a f]
  clojure.lang.ILookup
  (valAt [this k]
    (.valAt this k nil))
  (valAt [this k not-found]
    (case k
      :a a
      not-found))
  IBind
  (bind [this g]
    (mplus (g a) (fn [] (bind f g))))
  IMPlus
  (mplus [this fp]
    (Choice. a (fn [] (mplus (fp) f))))
  ITake
  (take* [this]
    (lazy-seq (cons (first a) (lazy-seq (take* f))))))

(extend-type nil
  IBind
  (bind [_ g] nil)
  IMPlus
  (mplus [_ f] (f))
  ITake
  (take* [_] '()))

(extend-type Object
  IMPlus
  (mplus [this f]
    (Choice. this f)))

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


The
Art
of the
Metaobject
Protocol
Gregor Kiczales
Jim des Rivières
Daniel G. Bobrow



... An exciting development within logic programming has been the realization that unification is just one instance of constraint solving ...

cKanren

cKanren

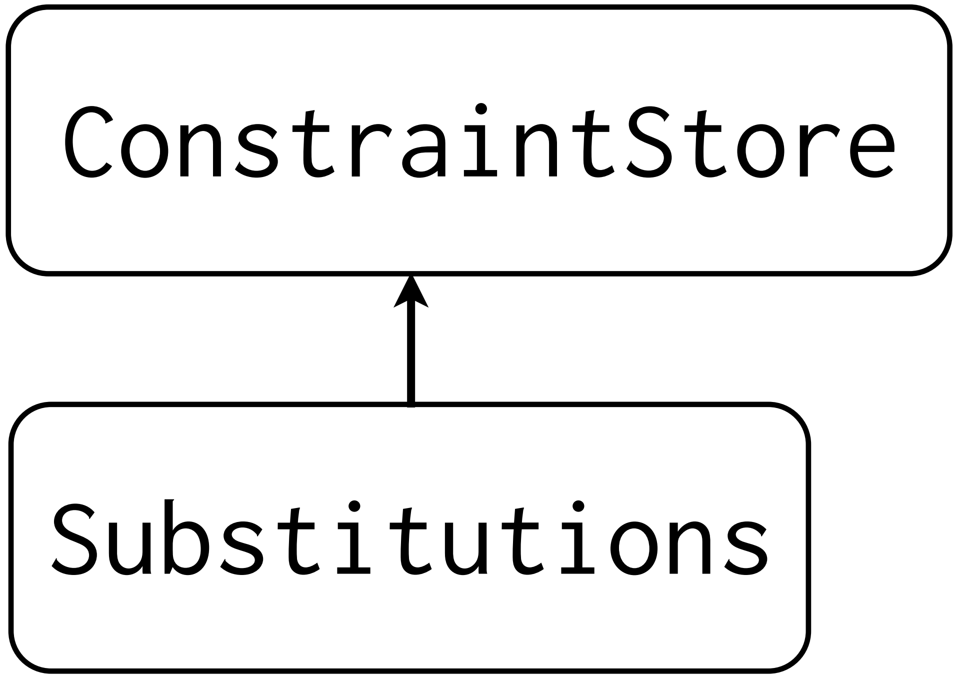
- Claire Alvis, William Byrd, Dan Friedman, et al Scheme Workshop 2011

cKanren

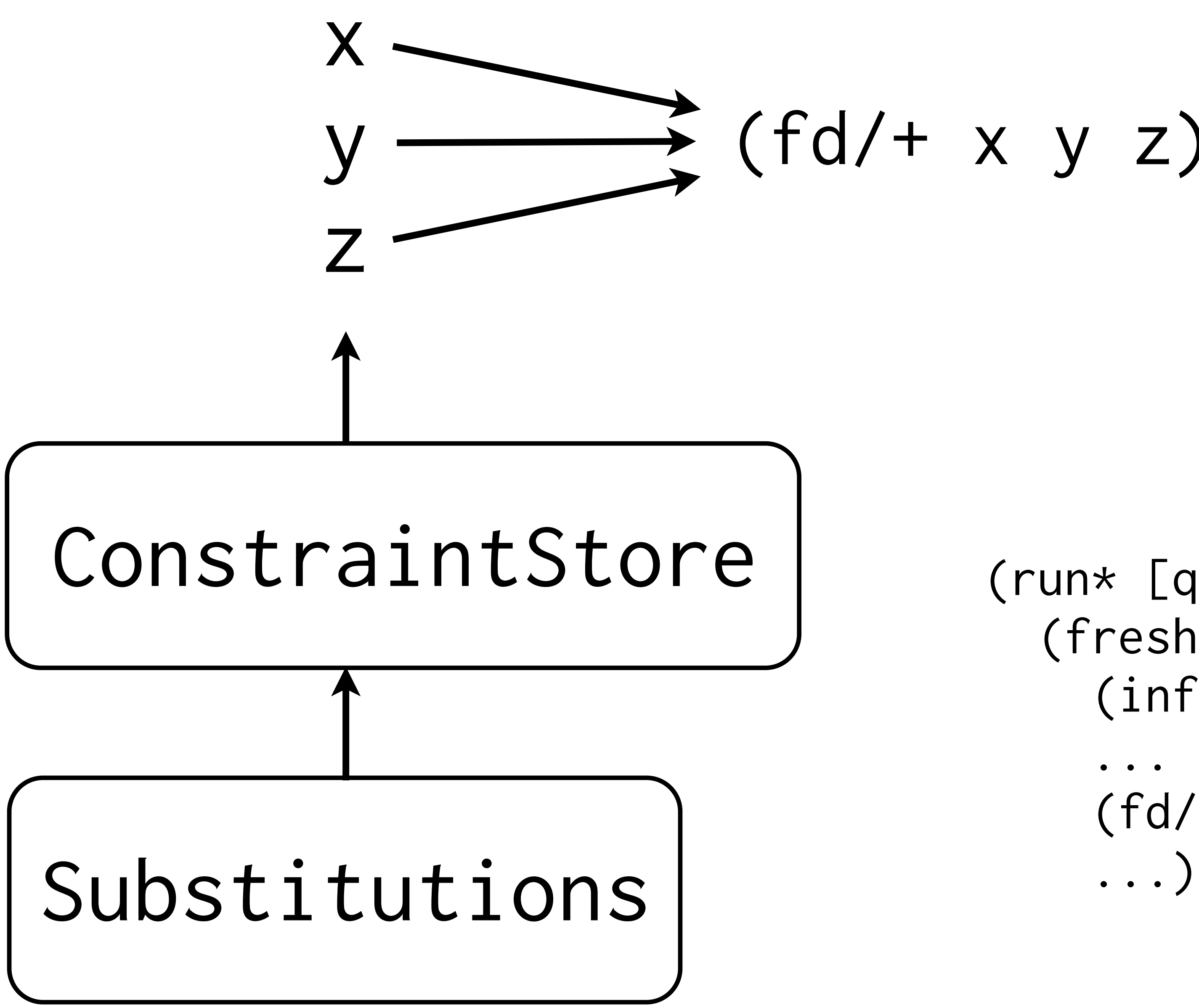
- Claire Alvis, William Byrd, Dan Friedman, et al Scheme Workshop 2011
- miniKanren extended to CLP(FD) & CLP(Tree)

cKanren

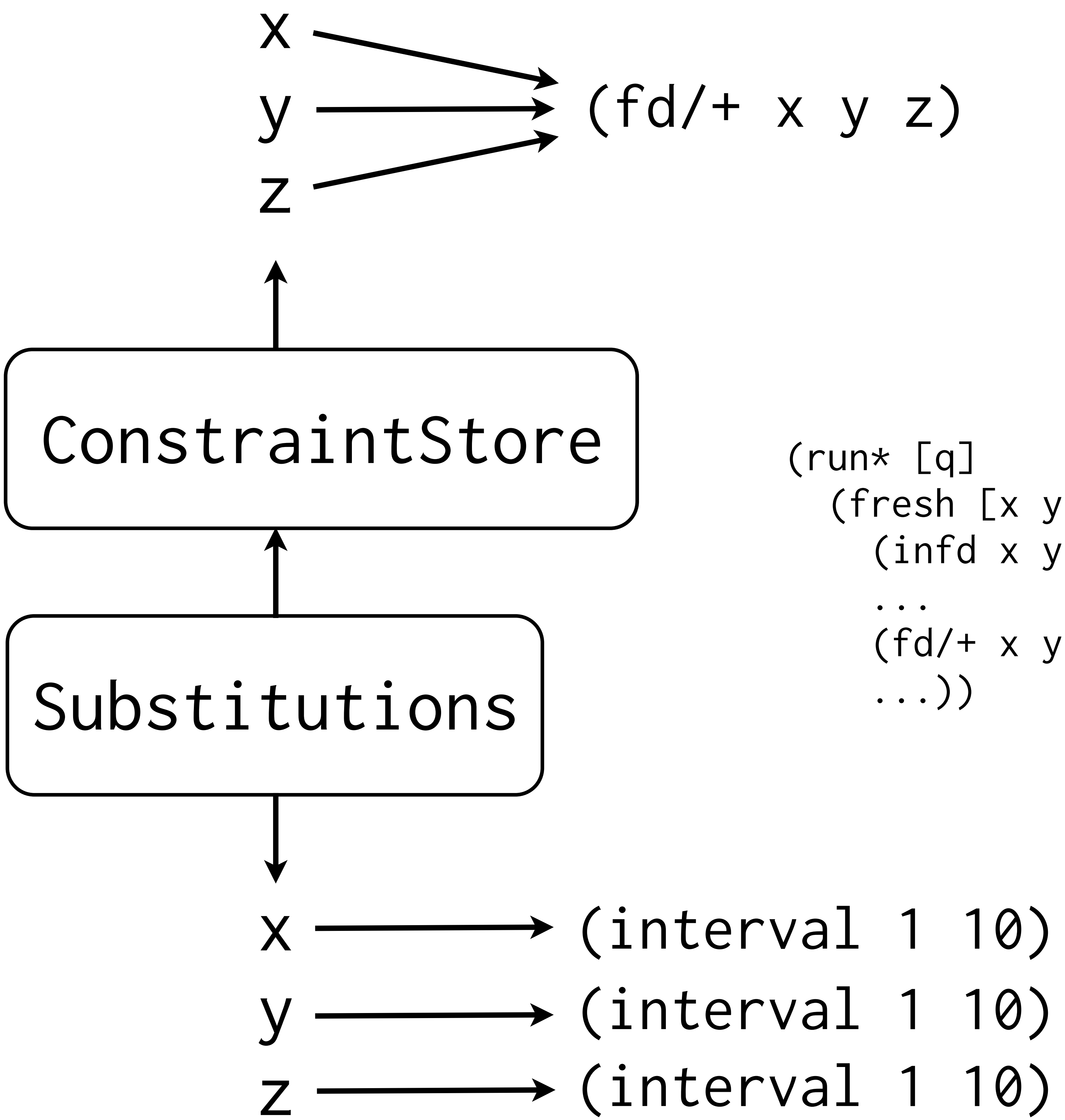
- Claire Alvis, William Byrd, Dan Friedman, et al Scheme Workshop 2011
- miniKanren extended to CLP(FD) & CLP(Tree)
- big idea - different constraint solvers should work well together!



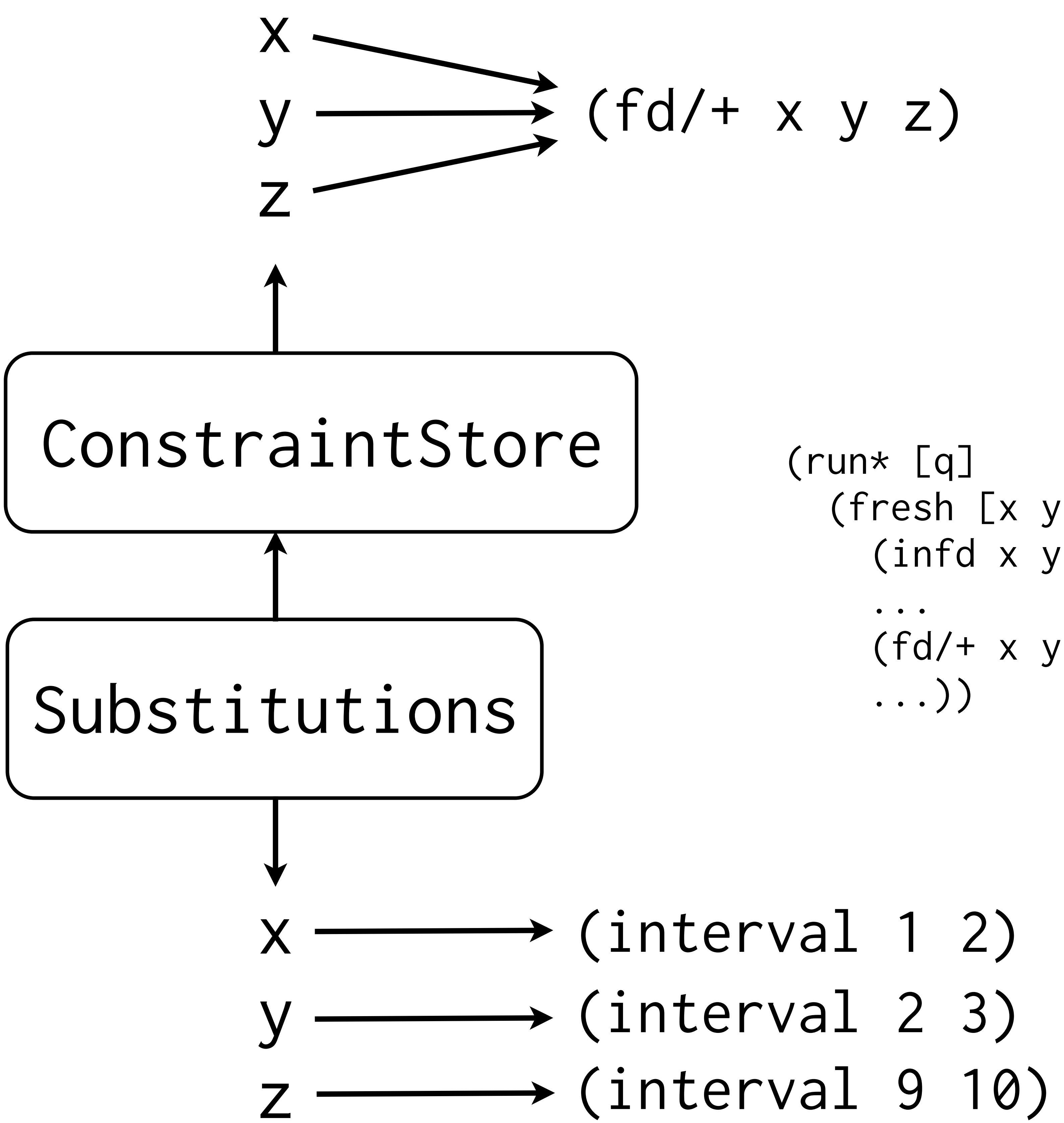

```
(run* [q]
  (fresh [x y z]
    (infd x y z (fd/interval 1 10))
    ...
    (fd/+ x y z)
    ...))
```




```
(run* [q]
  (fresh [x y z]
    (infd x y z (fd/interval 1 10))
    ...
    (fd/+ x y z)
    ...))
```




```
(run* [q]
  (fresh [x y z]
    (infd x y z (fd/interval 1 10))
    ...
    (fd/+ x y z)
    ...))
```




```
(run* [q]
  (fresh [x y z]
    (infd x y z (fd/interval 1 10))
    ...
    (fd/+ x y z)
    ...))
```

```
(define empty-s ())
```

```
(deftype Substitutions [s vs ts cs cq cqs oc _meta])
```

substitutions (PersistentHashMap)

```
[s vs ts cs cq cqs oc _meta]
```

vars altered during unification

(PersistentVector)

↓

```
[s vs ts cs cq cqs oc _meta]
```

goal tables

(mutable atom w/ PersistentHashMap)

↓

```
[s vs ts cs cq cqs oc _meta]
```

Constraint Store
(custom deftype)

↓

```
[s vs ts cs cq cqs oc _meta]
```

Constraint Queue
(PersistentVector)

↓

`[s vs ts cs cq cqs oc _meta]`

Constraint Queue Set

(PersistentVector)

$\downarrow$

```
[s vs ts cs cq cqs oc _meta]
```

Occurs check flag

(boolean)

↓

```
[s vs ts cs cq cqs oc _meta]
```

```
(deftype ConstraintStore [km cm cid running])
```

var $\rightarrow$ constraint id set

(PersistentHashMap)

$\downarrow$

`[km cm cid running]`

$$\text{constraint id} \rightarrow \text{constraint}$$
$$\text{(PersistentHashMap)}$$

$$\downarrow$$

`[km cm cid running]`

constraint id counter

(long)

$\downarrow$

`[km cm cid running]`

running constraint id

(long)

`[km cm cid running]`

```
(deftype ConstraintStore [km cm cid running]
  ...
  IConstraintStore
  (addc [this a c] ...)
  (updatec [this a c] ...)
  (remc [this a c] ...)
  (runc [this c state] ...)
  (constraints-for [this a x ws] ...)
  (migrate [this x root] ...))
```

```
(deftype ConstraintStore [km cm cid running]
  ...
  IConstraintStore
  (addc [this a c] ...)
  (updatec [this a c] ...)
  (remc [this a c] ...)
  (runc [this c state] ...)
  (constraints-for [this a x ws] ...)
  (migrate [this x root] ...))
```


$10,000

```
   S E N D
+  M O R E
----------
 M O N E Y
```

- Implementation in the cKanren paper uses long addition to solve efficiently

- Implementation in the cKanren paper uses long addition to solve efficiently
- Most users will not want to formulate their equations in this way

- Implementation in the cKanren paper uses long addition to solve efficiently
- Most users will not want to formulate their equations in this way
- I tried to write it in a more natural representation

```
(define cryptarithfd
  (lambda ()
    (run 1 (q)
      (fresh (s e n d m o r y
              p0 p1 p2 p3 p4 p5 p6 p7 p8 p9
              s0 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10)
        (== q `(,s ,e ,n ,d ,m ,o ,r ,y))
        (infd s e n d m o r y (range 0 9))
        (infd p0 p3 p7 (range 0 9000))
        (infd p1 p4 p8 (range 0 900))
        (infd p2 p5 p9 (range 0 90))
        (infd p6 (range 0 90000))
        (infd s0 (range 0 9900))
        (infd s1 (range 0 9990))
        (infd s2 (range 0 9999))
        (infd s3 (range 0 18999))
        (infd s4 (range 0 19899))
        (infd s5 (range 0 19989))
        (infd s6 (range 0 19998))
        (infd s7 (range 0 99000))
        (infd s8 (range 0 99900))
        (infd s9 (range 0 99990))
        (infd s10 (range 0 99999))
        (distinctfd q)
        (=/=fd m 0) (=/=fd s 0)
        (timesfd 1000 s p0) (timesfd 100 e p1) (timesfd 10 n p2)
        (timesfd 1000 m p3) (timesfd 100 o p4) (timesfd 10 r p5)
        (timesfd 10000 m p6) (timesfd 1000 o p7) (timesfd 100 n p8) (timesfd 10 e p9)
        (plusfd p0 p1 s0) (plusfd s0 p2 s1) (plusfd s1 d s2)
        (plusfd s2 p3 s3) (plusfd s3 p4 s4) (plusfd s4 p5 s5) (plusfd s5 e s6)
        (plusfd p6 p7 s7) (plusfd s7 p8 s8) (plusfd s8 p9 s9) (plusfd s9 y s10)
        (=fd s6 s10)))))
```

$>$42 seconds (Petite Chez)

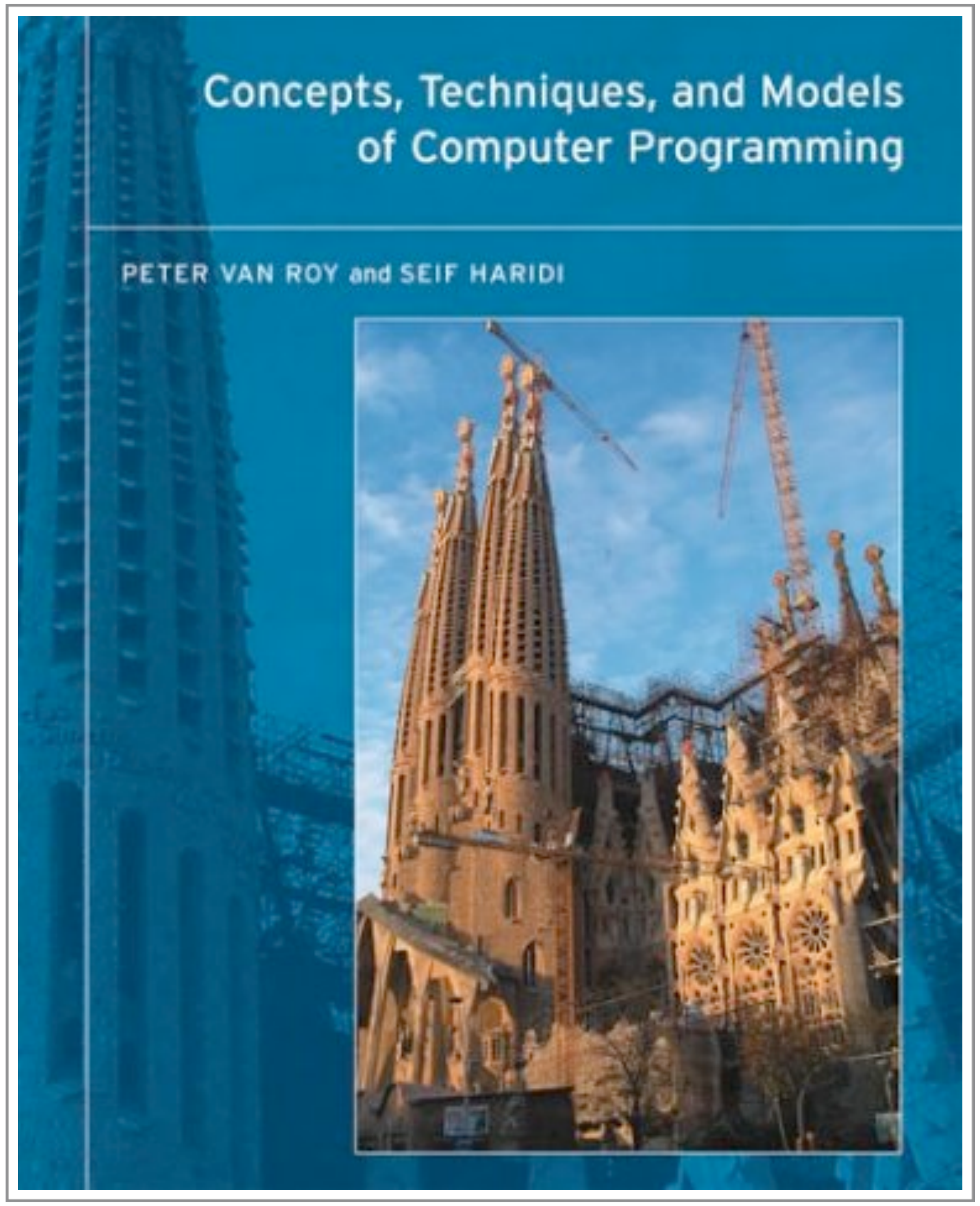
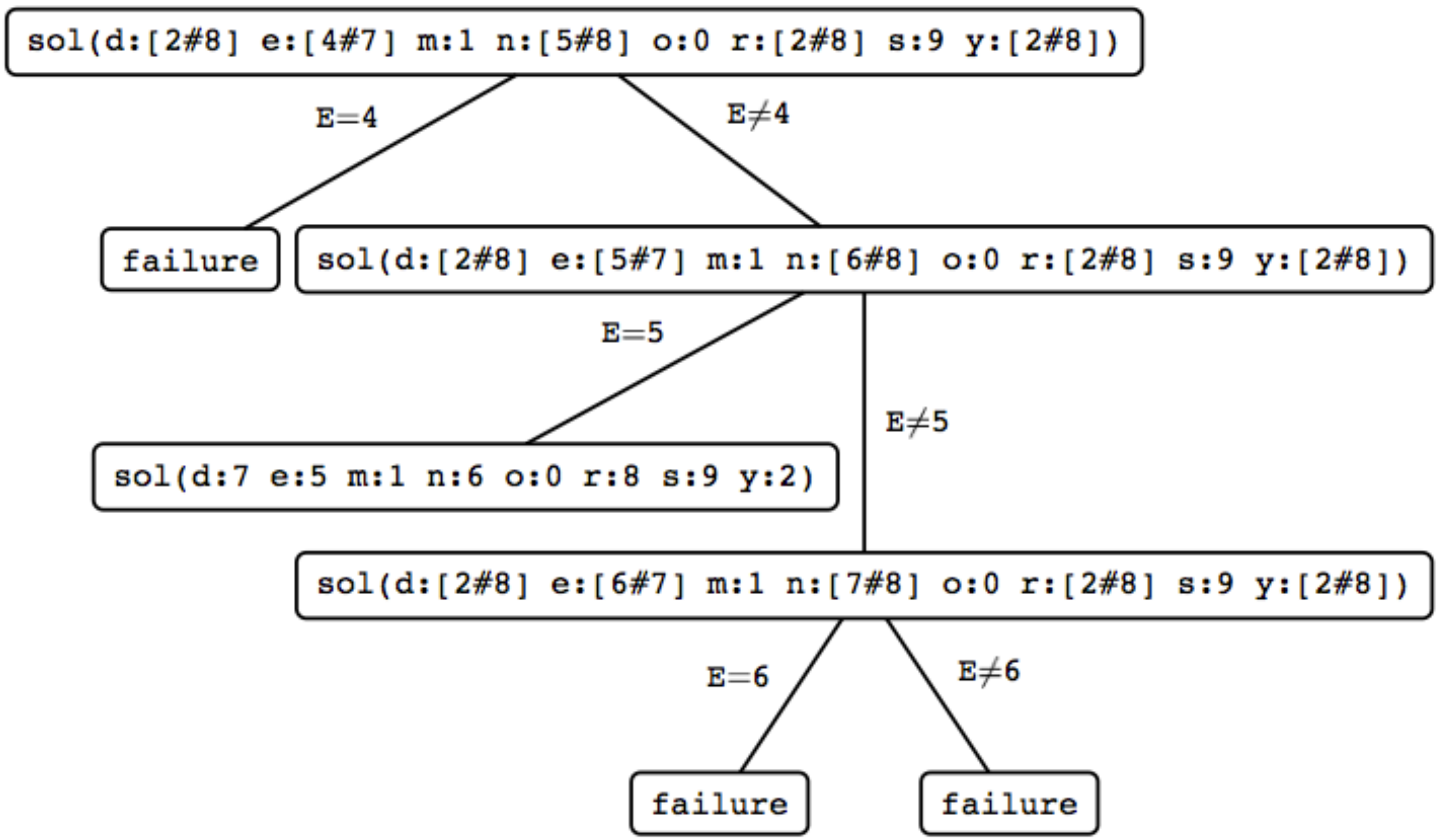

Concepts, Techniques, and Models
of Computer Programming
PETER VAN ROY and SEIF HARIDI

Figure 3.1 A script for the Send More Money Puzzle.

```
proc {Money Root}
   S E N D M O R Y
in
   Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)      % 1
   Root ::: 0#9                                     % 2
   {FD.distinct Root}                               % 3
   S \=: 0                                          % 4
   M \=: 0
                1000*S + 100*E + 10*N + D           % 5
   +            1000*M + 100*O + 10*R + E
   =: 10000*M + 1000*O + 100*N + 10*E + Y
   {FD.distribute ff Root}
end
```

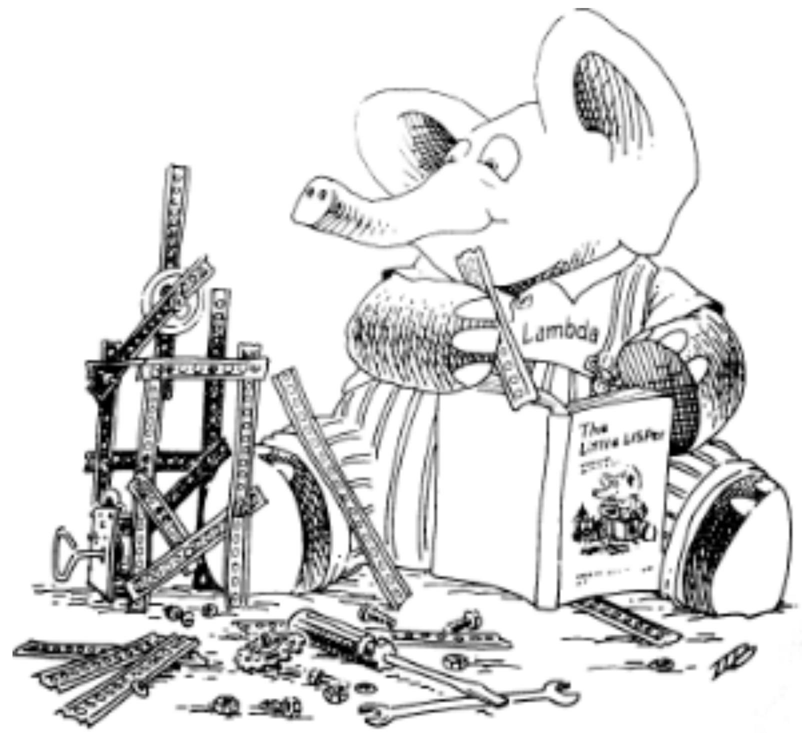
Figure 3.2 The search tree explored by `Money`.



- cKanren only runs interval constraints once!

- cKanren only runs interval constraints once!

- To avoid an infinite loop constraints are taken out of the store when run, and added back only when triggered constraints have completed



What if ...

- When a constraint runs, it may trigger other constraints

- When a constraint runs, it may trigger other constraints
- Queue them

- When a constraint runs, it may trigger other constraints
- Queue them
- Pop/run next constraint in queue, which may queue the original constraint again, this is OK!

- When a constraint runs, it may trigger other constraints
- Queue them
- Pop/run next constraint in queue, which may queue the original constraint again, this is OK!
- Fixpoint, no constraint could change any domains

Running Queued

`(fd/+ x y z)`

Running	Queued
`(fd/+ x y z)`	`(fd/* a b y)`
	`(fd/+ c d y)`

Running Queued

```
(fd/* a b y)
(fd/+ c d y)
```

Running Queued

`(fd/* a b y)` `(fd/+ c d y)`

Running	Queued
`(fd/* a b y)`	`(fd/+ c d y)`
	`(fd/+ x y z)`

```
([clojure.core.logic.fd/* [100 <lvar:e6524> <lvar:G__59086535>]]
 [clojure.core.logic.fd/* [10 <lvar:e6524> <lvar:G__59246551>]]
 [clojure.core.logic.fd/+ [<lvar:G__59176544> <lvar:e6524> <lvar:G__59166543>]]
 [clojure.core.logic.fd/+ [<lvar:d6526> <lvar:G__59126539> <lvar:G__59116538>]]
 [clojure.core.logic.fd/* [100 <lvar:n6525> <lvar:G__59226549>]]
 [clojure.core.logic.fd/* [10 <lvar:n6525> <lvar:G__59106537>]]
 [clojure.core.logic.fd/* [1000 <lvar:s6523> <lvar:G__59066533>]]
 [clojure.core.logic.fd/* [1000 <lvar:o6528> <lvar:G__59206547>]]
 [clojure.core.logic.fd/* [100 <lvar:o6528> <lvar:G__59156542>]]
 [clojure.core.logic.fd/+ [<lvar:G__59246551> <lvar:y6530> <lvar:G__59236550>]]
 [clojure.core.logic.fd/* [10 <lvar:r6529> <lvar:G__59176544>]]
 [clojure.core.logic.fd/+ [<lvar:G__59136540> <lvar:G__59146541> <lvar:G__59126539>]]
 [clojure.core.logic.fd/* [10000 <lvar:m6527> <lvar:G__59186545>]])
```

578 calls to `run-constraint`

22 calls to `map-sum`

```
(defn cryptarithfd-1 []
  (run-nc* [s e n d m o r y :as q]
    (fd/in s e n d m o r y (fd/interval 0 9))
    (fd/distinct q)
    (fd/!= m 0) (fd/!= s 0)
    (fd/eq
      (=             (+ (* 1000 s) (* 100 e) (* 10 n) d
                        (* 1000 m) (* 100 o) (* 10 r) e)
         (+ (* 10000 m) (* 1000 o) (* 100 n) (* 10 e) y)))))
```

Figure 3.1 A script for the Send More Money Puzzle.

```
proc {Money Root}
   S E N D M O R Y
in
   Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)       % 1
   Root ::: 0#9                                       % 2
   {FD.distinct Root}                                 % 3
   S \=: 0                                            % 4
   M \=: 0
                1000*S + 100*E + 10*N + D             % 5
   +            1000*M + 100*O + 10*R + E
   =: 10000*M + 1000*O + 100*N + 10*E + Y
   {FD.distribute ff Root}
end
```

$\sim$15ms on Clojure JVM

- core.logic `SEND+MORE=MONEY` still involves too much search

- core.logic `SEND+MORE=MONEY` still involves too much search
- We don't have equation level constraints

- core.logic `SEND+MORE=MONEY` still involves too much search
- We don't have equation level constraints
- We should see precisely the amount of search as demonstrated by Mozart/OZ

Anatomy of a Constraint

- Like goals, constraints are just closures. They return a reified object that implements IFn and satisfies a number of custom protocols (subject to change)

- Like goals, constraints are just closures. They return a reified object that implements IFn and satisfies a number of custom protocols (subject to change)

- We can support much of the necessary “meta” behavior of cKanren constraints without macros

- Like goals, constraints are just closures. They return a reified object that implements IFn and satisfies a number of custom protocols (subject to change)

- We can support much of the necessary “meta” behavior of cKanren constraints without macros

  - yet leave the door open to use macros to provide a friendlier interface

```clojure
(defn -domc [x]
  (reify
    IEnforceableConstraint
    clojure.lang.IFn
    (invoke [this s]
      (when (member? (get-dom s x) (walk s x))
        (rem-dom s x ::l/fd)))
    IConstraintOp
    (rator [_] `domc)
    (rands [_] [x])
    IRelevant
    (-relevant? [this s]
      (not (nil? (get-dom s x))))
    IRunnable
    (runnable? [this s]
      (not (lvar? (walk s x))))
    IConstraintWatchedStores
    (watched-stores [this] #{::l/subst})))
```

Invoke like a normal function

```
(defn -domc [x]
  (reify
    IEnforceableConstraint
    clojure.lang.IFn
    (invoke [this s]
      (when (member? (get-dom s x) (walk s x))
        (rem-dom s x ::l/fd)))
    IConstraintOp
    (rator [_] `domc)
    (rands [_] [x])
    IRelevant
    (-relevant? [this s]
      (not (nil? (get-dom s x))))
    IRunnable
    (runnable? [this s]
      (not (lvar? (walk s x))))
    IConstraintWatchedStores
    (watched-stores [this] #{::l/subst})))
```

What vars are associated with this constraint? →

```
(defn -domc [x]
  (reify
    IEnforceableConstraint
    clojure.lang.IFn
    (invoke [this s]
      (when (member? (get-dom s x) (walk s x))
        (rem-dom s x ::l/fd)))
    IConstraintOp
    (rator [_] `domc)
    (rands [_] [x])
    IRelevant
    (-relevant? [this s]
      (not (nil? (get-dom s x))))
    IRunnable
    (runnable? [this s]
      (not (lvar? (walk s x))))
    IConstraintWatchedStores
    (watched-stores [this] #{::l/subst})))
```

```clojure
(defn -domc [x]
  (reify
    IEnforceableConstraint
    clojure.lang.IFn
    (invoke [this s]
      (when (member? (get-dom s x) (walk s x))
        (rem-dom s x ::l/fd)))
    IConstraintOp
    (rator [_] `domc)
    (rands [_] [x])
    IRelevant
    (-relevant? [this s]
      (not (nil? (get-dom s x))))
    IRunnable
    (runnable? [this s]
      (not (lvar? (walk s x))))
    IConstraintWatchedStores
    (watched-stores [this] #{::l/subst})))
```

What var changes do we care about?

First run

```
(defn cgoal [c]
  (reify
    clojure.lang.IFn
    (invoke [_ a]
      (if (runnable? c a)
        (when-let [a (c a)]
          (if (and (irelevant? c) (relevant? c a))
            ((addcg c) a)
            a))
        ((addcg c) a)))
    ...))
```

```
(defn cgoal [c]
  (reify
    clojure.lang.IFn
    (invoke [_ a]
      (if (runnable? c a)
        (when-let [a (c a)]
          (if (and (irelevant? c) (relevant? c a))
            ((addcg c) a)
            a))
        ((addcg c) a)))
    ...))
```

vroom vroom?

```
(defn cgoal [c]
  (reify
    clojure.lang.IFn
    (invoke [_ a]
      (if (runnable? c a)
        (when-let [a (c a)]
          (if (and (irelevant? c) (relevant? c a))
            ((addcg c) a)
            a))
        ((addcg c) a)))
    ...))
```

did it work?

```
(defn cgoal [c]
  (reify
    clojure.lang.IFn
    (invoke [_ a]
      (if (runnable? c a)
        (when-let [a (c a)]
          (if (and (irelevant? c) (relevant? c a))
            ((addcg c) a)
            a))
        ((addcg c) a)))
    ...))
```

are we there yet?

```
(defn cgoal [c]
  (reify
    clojure.lang.IFn
    (invoke [_ a]
      (if (runnable? c a)
        (when-let [a (c a)]
          (if (and (irelevant? c) (relevant? c a))
            ((addcg c) a)
            a))
        ((addcg c) a)))
    ...))
```

Add to constraint store!

Subsequent runs

```clojure
(defn run-constraint [c]
  (fn [a]
    (if (or (not (irelevant? c)) (relevant? c a))
      (if (runnable? c a)
        ((composeg* (runcg c) c (stopcg c)) a)
        a)
      ((remcg c) a))))
```

```
(defn run-constraint [c]
  (fn [a]
    (if (or (not (irelevant? c)) (relevant? c a))
      (if (runnable? c a)
        ((composeg* (runcg c) c (stopcg c)) a)
        a)
      ((remcg c) a))))
```

```
(defn run-constraint [c]
  (fn [a]
    (if (or (not (irelevant? c)) (relevant? c a))
      (if (runnable? c a)
        ((composeg* (runcg c) c (stopcg c)) a)
        a)
      ((remcg c) a))))
```

```
(defn run-constraint [c]
  (fn [a]
    (if (or (not (irelevant? c)) (relevant? c a))
      (if (runnable? c a)
        ((composeg* (runcg c) c (stopcg c)) a)
        a)
      ((remcg c) a))))
```

```
(defn run-constraint [c]
  (fn [a]
    (if (or (not (irelevant? c)) (relevant? c a))
      (if (runnable? c a)
        ((composeg* (runcg c) c (stopcg c)) a)
        a)
      ((remcg c) a))))
```

Complications

LVars

LVars

- Where to store domain information?

LVars

- Where to store domain information?
  - Originally stored directly in substitution

LVars

- Where to store domain information?
  - Originally stored directly in substitution
  - Trouble, could not run unification without triggering constraints

SubstValue

SubstValue

- `LVar`s can point to normal Clojure values

`SubstValue`

- `LVar`s can point to normal Clojure values
- *Or* be unbound and point to domain information, `SubstValue`

SubstValue

- `LVar`s can point to normal Clojure values
- *Or* be unbound and point to domain information, `SubstValue`
- `walk` (lvar lookup) knows not to return `SubstValue`

`SubstValue`

- `LVar`s can point to normal Clojure values
- *Or* be unbound and point to domain information, `SubstValue`
- `walk` (lvar lookup) knows not to return `SubstValue`
  - so need to use `root-val` for access to `SubstValue`

```
(walk [this v]
  (if (bindable? v)
    (loop [lv v [v vp :as me] (find s v)]
      (cond
        (nil? me) lv

        (not (bindable? vp))
        (if (subst-val? vp)
          (let [sv (:v vp)]
            (if (= sv ::unbound)
              (with-meta v (assoc (meta vp) ::unbound true))
              sv))
          vp)

        :else (recur vp (find s vp))))
    v))
```

Note, not calls to `lvar?` :)

```
(walk [this v]
  (if (bindable? v)
    (loop [lv v [v vp :as me] (find s v)]
      (cond
        (nil? me) lv

        (not (bindable? vp))
        (if (subst-val? vp)
          (let [sv (:v vp)]
            (if (= sv ::unbound)
              (with-meta v (assoc (meta vp) ::unbound true))
              sv))
          vp)

        :else (recur vp (find s vp))))
    v))
```

```
(walk [this v]
  (if (bindable? v)
    (loop [lv v [v vp :as me] (find s v)]
      (cond
        (nil? me) lv

        (not (bindable? vp))
        (if (subst-val? vp)
          (let [sv (:v vp)]
            (if (= sv ::unbound)
              (with-meta v (assoc (meta vp) ::unbound true))
              sv))
          vp)

        :else (recur vp (find s vp))))
    v))
```

```
(walk [this v]
  (if (bindable? v)
    (loop [lv v [v vp :as me] (find s v)]
      (cond
        (nil? me) lv

        (not (bindable? vp))
        (if (subst-val? vp)
          (let [sv (:v vp)]
            (if (= sv ::unbound)
              (with-meta v (assoc (meta vp) ::unbound true))
              sv))
          vp)

        :else (recur vp (find s vp))))
    v))
```

Var aliasing

Var aliasing

- Must be careful that the constraint store only stores *roots*

Var aliasing

- Must be careful that the constraint store only stores *roots*
- Why? If y $\rightarrow$ x, if constraint applied to y, we really want to constrain x!

Var aliasing

- Must be careful that the constraint store only stores *roots*
- Why? If y $\rightarrow$ x, if constraint applied to y, we really want to constrain x!
- Unification may mean we need to re-root

Var aliasing

- Must be careful that the constraint store only stores *roots*
- Why? If y $\rightarrow$ x, if constraint applied to y, we really want to constrain x!
- Unification may mean we need to re-root
  - and migrate constraints

```
{x <interval 1 3>,
 y <interval 2 4>}
```

```
{x <interval 2 3>,
 y x}
```

AoMCLP

AoMCLP

- Good protocols (not there yet)

AoMCLP

- Good protocols (not there yet)
  - protocols should allow further optimizations by constraint solver authors - are we exposing enough?

AoMCLP

- Good protocols (not there yet)
  - protocols should allow further optimizations by constraint solver authors - are we exposing enough?
- αKanren is a first step, each new constraint domain, more confidence

Future directions

Future directions

- CLP(HashMap|Set), allow any hash/set data structure to participate

Future directions

- CLP(HashMap|Set), allow any hash/set data structure to participate
- Polymorphic constraints and goals

Future directions

- CLP(HashMap|Set), allow any hash/set data structure to participate
- Polymorphic constraints and goals
- CLP(Prob)

Future directions

- CLP(HashMap|Set), allow any hash/set data structure to participate
- Polymorphic constraints and goals
- CLP(Prob)
- Making tabling way cooler

Future directions

- CLP(HashMap|Set), allow any hash/set data structure to participate
- Polymorphic constraints and goals
- CLP(Prob)
- Making tabling way cooler
- External solvers (GeCode, JaCoP)?

Future directions

- CLP(HashMap|Set), allow any hash/set data structure to participate
- Polymorphic constraints and goals
- CLP(Prob)
- Making tabling way cooler
- External solvers (GeCode, JaCoP)?
- ... what else?

Thanks!

Questions?