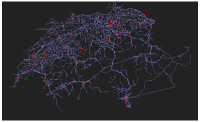
Clojure und core.logic

...hello to the world of logic programming

christian.meichsner@xelog.com

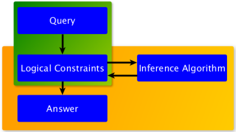
25. Februar 2014

Visual Index



Clojure and me

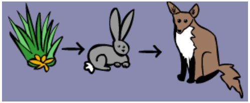
Dark is life, dark is death



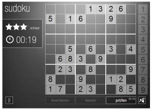
What is logic programming

LISP & Clojure Primer

Logic programming using Clojure and core.logic



Hello World: model food chains



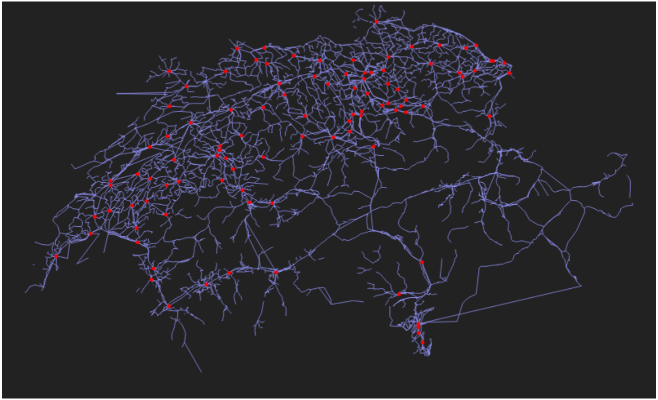
Sudoku in 30 LoC

Logic programming - mission critical

Clojure and me

Swiss public transportion service - Contractual network & pricing models

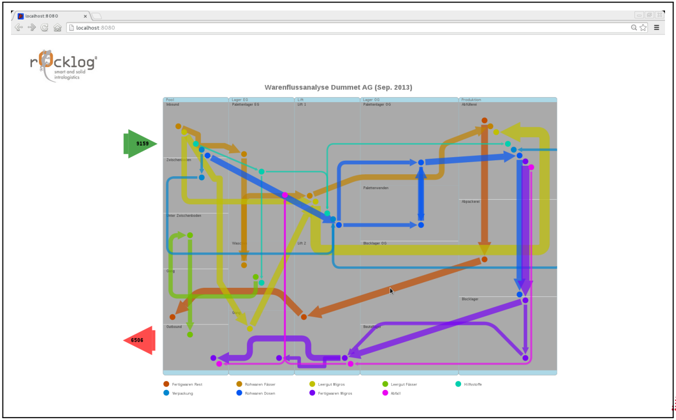
- network design & graph search algorithms
- pricing models & impact analysis for transport service providers



Clojure and me

Goods flow analysis tool

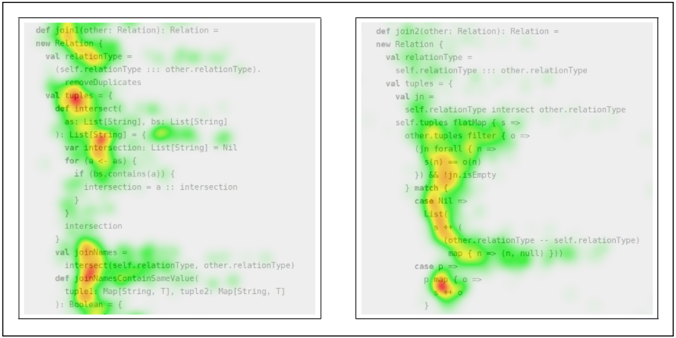
- visualizing goods flows
- optimizing transportation capacities in warehouses (stackers, lifts)
- genetic algorithms apply core.logic



JAVA USER GROUP CH

Clojure and me

Are computer languages improving?



1

[1] Gilles Dubochet (2009): Computer Code as Medium for Human Communication: Are Computer Languages Improving

Imperative Programming

(or ...) Dark is life, dark is death

(Imperative) Velocireptor

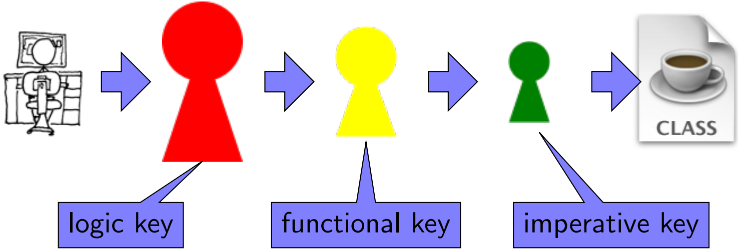
(The replaceable) you

Imperative Programming

Dark is life, dark is death - How is that?

Imperative Programming

The three keys needed...



Imperative Programming

The three keys needed...

logic key

functional key

imperative key

- constraints
- axioms
- facts
- relations
- conjuction
- disjunction
- (finite) domains
- algebra of sets

Imperative Programming

The three keys needed...

logic key	functional key	imperative key

logic key

- constraints
- axioms
- facts
- relations
- conjuction
- disjunction
- (finite) domains
- algebra of sets

functional key

- (partial) functions
- generic datastructures
- generic sequence handling
- recursion
- identity
- state
- pattern matching
- high-level concurreny

imperative key

JAVA USER GROUP CH

Imperative Programming

The three keys needed...

logic key | functional key | imperative key

- constraints
- axioms
- facts
- relations
- conjuction
- disjunction
- (finite) domains
- algebra of sets

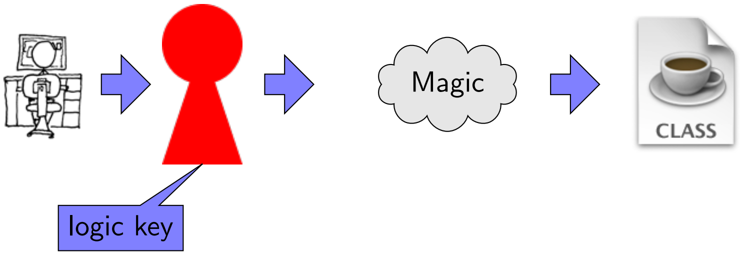
- (partial) functions
- generic datastructures
- generic sequence handling
- recursion
- identity
- state
- pattern matching
- high-level concurreny

- classes
- functions
- instances
- for(i in I)
- if then (else)
- switch
- @Annotations
- immutable datastructures
- mutable datastructures
- setter / getter
- mutexes / semaphores
- monitor
- Big Decimal vs. Long

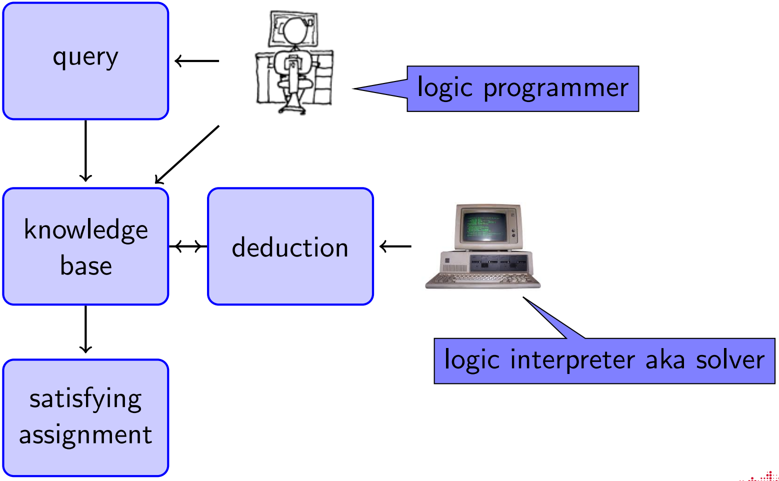
- enums
- introspection
- generics
- type erasure
- thread
- timer
- timertask
- future
- threadpool
- fork-join
- a++
- ++a
- operator precendence
- operator associativity

Logic Programming

Just one key is needed...

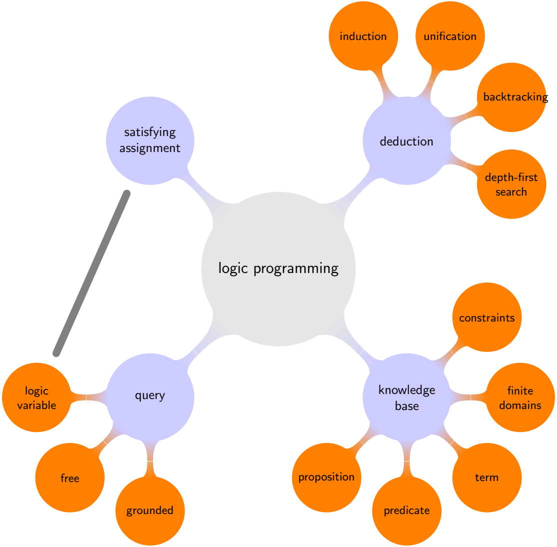


What is logic programming?



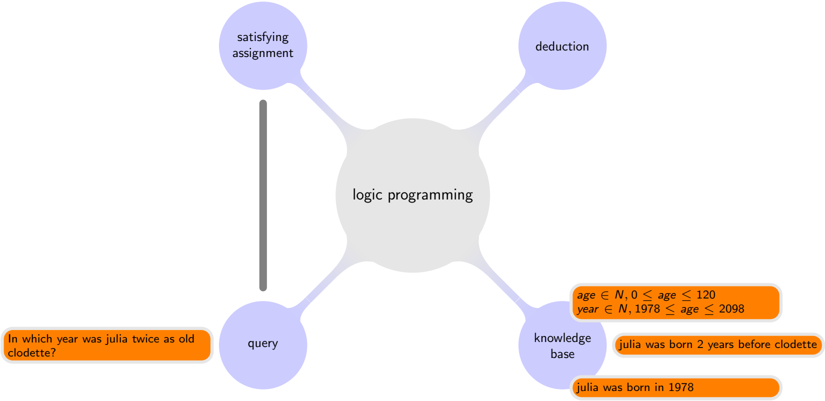
What is logic programming?

semantic elements



What is logic programming?

abstract ... concrete



LISP & Clojure Primer
LISP Primer #1

- **LIS**T **P**rocessing, invented by John McCarthy in 1958 at MIT

- fully parenthesized prefix notation
- syntax elements countable with two hands

```
(...)           ;;list
'...            ;;quote
:age            ;;keyword
"..."           ;;string literal
3 3.1 1/3       ;;numeric literals
[...]           ;;vector
#{1 2}          ;;set
{:age 1}        ;;map
@my-future      ;;dereferencing identities and futures
```

LISP & Clojure Primer

LISP Primer #2

- everything is a list

```
(inc
  (+ 1
     (* 2 2)))
```

- homoiconic language (code-is-data)

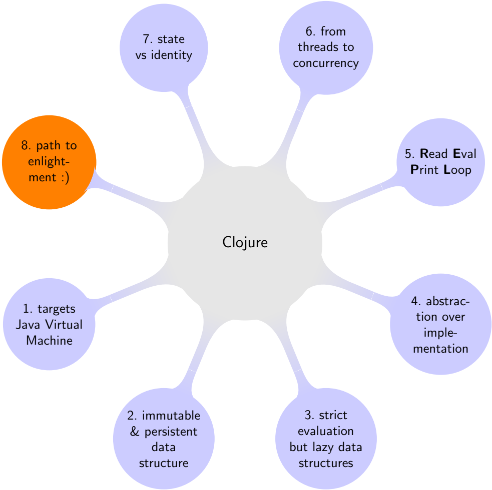
```
user=> (class '(+ 1 2 3 4 5))
clojure.lang.PersistentList
user=>
```

- programmable programming language - hygenic macros

```
(defmacro dyn-for [xs]
  `(let [sym-index# (zipmap (repeatedly (fn [](gensym))) ~xs)
         keyvals# (reduce #(conj % (first %2) (second %2)) [] sym-index#)
         fd# (list 'for keyvals# (vec (reverse (map first sym-index#))))]
     (eval fd#)))
```

LISP & Clojure Primer

Clojure rocks!



Logic programming using Clojure and core.logic

structure of a logic programm

Starts the logical interpreter

logic variable

goal 1

goal 2

```
(run* [q]
      (membero q [1 2 3])
      (membero q [3 4 5]))
```

- **run*** returns all satisfying assignments
- a **logic variable** can take several values, but just one at a time
- **goals** express the knowledgebase. a goal succeeds, or does not. all goals must succeed in order to provide a satisfying assignment to the query.

Logic programming using Clojure and core.logic

run*

```
(run* [q r s]
      (membero q [1 2 3])
      (membero r [2 3 4])
      (membero s [3 4 5]))
```

- **run*** can refer to more than one lvar
- if so, a list of vectors is returned

Logic programming using Clojure and core.logic

==

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

- **==** is the most elementary logic operation, called unification
- (== q 1) succeeds iff q can be associated to 1 ... and associates q to 1 :)

Logic programming using Clojure and core.logic

conde

```
(run* [q]
      (conde
        [(== q 1)]
        [(== q "zwei")]))
```

- **conde** is like OR
- (conde g1 ... gn) succeeds, iff one of the goals g1 ... gn succeeds

Logic programming using Clojure and core.logic

!=

```
(run* [q]
      (conde
        [(== q 1)]
        [(== q 2)])
      (!= q 2))
```

- **!=** is called disunification
- (!= q a) succeeds and ensures that q is never associated to a

Logic programming using Clojure and core.logic

membero

```
(run* [s p o]
      (membero s [:mother :child])
      (membero o [:mother :child])
      (membero p [:loves :has])
      (!= s o))
```

- (**membero** x l) constraints x to be an element of l

Logic programming using Clojure and core.logic

distincto

```
(run* [s p o]
      (membero s [:mother :child])
      (membero o [:mother :child])
      (membero p [:loves :has])
      (distincto [s o]))
```

- (**distincto** [x1 ... xn]) constraints x1 ... xn to be disjunct

```
(run* [s p o]
      (everyg #(membero % [:mother :child]
                         [s o]))
      (membero p [:loves :has])
      (distincto [s o]))
```

- (**everyg** f [x1 ... xn]) succeeds, iff goals f(x1) ... f(xn) succeed

Logic programming using Clojure and core.logic

fresh

```
(run* [languages]
      (fresh [a b c d]
             (== a "romansh")
             (== b "italian")
             (== c "french")
             (== d "german")
             (== languages [a b c d])))

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

- (**fresh** [q1 ... qn] g1 ... gn) creates a new lexical scope and fresh lvars q1 ... qn and succeeds, iff goals g1 ... gn succeed

Logic programming using Clojure and core.logic

Constraint logic programming over finite domains CLP(FD)

```
(run* [q]
      (fd/in q (fd/interval 0 9)))
```

- **fd/interval** defines a finite domain over positive integers
- (**fd/in** q1 ... qn d) constraints lvar q1 ... qn to be in finite domain d

Logic programming using Clojure and core.logic

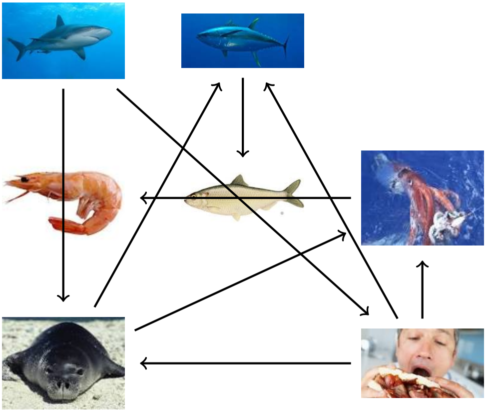
Constraint logic programming over finite domains CLP(FD)

```
(run* [q]
      (fresh [a b]
        (fd/in a b (fd/interval 0 9))
        (fd/+ a b 10)
        (== q [a b])))
```

- namespace **clojure.core.logic.fd** (here fd) offers operators to check simple arithmetic constraints

Logic programming using Clojure and core.logic

Modeling food chains using relational programming



Logic programming using Clojure and core.logic

Modeling food chains using relational programming

```
(facts/db-rel eats creature1 creature2)

(def factbase
  (facts/db
    [eats :shark :seal]
    [eats :seal :tuna]
    [eats :tuna :herring]
    [eats :human :seal]
    [eats :human :tuna]
    [eats :human :calamar]
    [eats :shark :human]
    [eats :seal :calamar]
    [eats :calamar :prawn]))

(facts/with-db
  factbase
  (run* [q]
    (fresh [x y z]
      (eats :shark x)
      (eats x y)
      (eats y z)
      (== q [:shark x y z]))))
```

- returns all food chains of length 4 with the shark being the top-notch

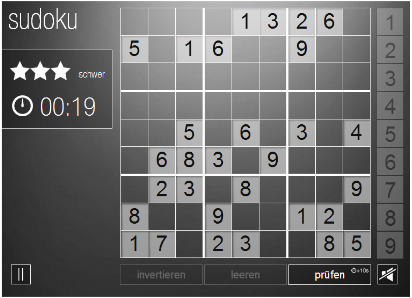
Sudoku in 30 LoC

[2]

[2] http://www.nzz.ch/lebensart/spiele/sudoku/

Sudoku in 30 LoC

black-box




```
(sodoku
   [0 0 0 0 1 3 2 6 0
    5 0 1 6 0 0 9 0 0
    0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0
    0 0 5 0 6 0 3 0 4
    0 6 8 3 0 9 0 0 0
    0 2 3 0 8 0 0 0 9
    8 0 0 9 0 0 1 2 0
    1 7 0 2 3 0 0 8 5])
```

JAVA USER GROUP CH

Sudoku in 30 LoC

(allmost) done

```
(defn sodoku [hints]
  (let [board (repeatedly 81 lvar)
        rows (->> board (partition 9) (map vec) (into []))
        cols (apply map vector rows)
        squares (for [x (range 0 9 3)
                      y (range 0 9 3)]
                  (square rows x y))]
    (run 1 [q]
      (== q board)
      (everyg #(fd/in % (fd/domain 1 2 3 4 5 6 7 8 9)) board)
      (init-board board hints)
      (everyg fd/distinct rows)
      (everyg fd/distinct cols)
      (everyg fd/distinct squares))))
```

- big-deal: constraint numbers in rows, columns and 3x3 squares to be distinct
- square ?
- init-board ?

Sudoku in 30 LoC

(allmost) done

```
(defn sodoku [hints]
  (let [board (repeatedly 81 lvar)
        rows (->> board (partition 9) (map vec) (into []))
        cols (apply map vector rows)
        squares (for [x (range 0 9 3)
                      y (range 0 9 3)]
                  (square rows x y))]
    (run 1 [q]
      (== q board)
      (everyg #(fd/in % (fd/domain 1 2 3 4 5 6 7 8 9)) board)
      (init-board board hints)
      (everyg fd/distinct rows)
      (everyg fd/distinct cols)
      (everyg fd/distinct squares))))
```

- big-deal: constraint numbers in rows, columns and 3x3 squares to be distinct
- square ?
- init-board ?

Sudoku in 30 LoC

```
(defn square [rows x y]
  (for [x (range x (+ x 3))
        y (range y (+ y 3))]
    (get-in rows [x y])))
```

- list comprehension to get a certain 3x3 square

Sudoku in 30 LoC

```
(defn init-board [vars hints]
  ;;check for emptiness
  (if
    (seq vars)
    (let [hint (first hints)]
      (all
        (if
          (zero? hint)
          succeed
          ;;else
          (== (first vars) hint))
        (init-board (next vars) (next hints))))
    ;;else - emptiness
    succeed))
```

- init lvars to be either grounded (hint) or free

Logic programming - mission critical

Ouch!

```c
volatile int a;
void baz(void) {
  int i;
  for (i=0; i<3; i++)
  {
    a += 7;
  }
}
```

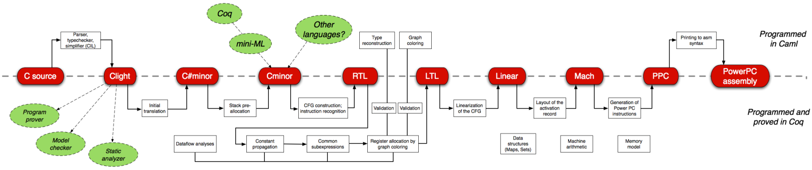
```asm
baz:
movl   a, %eax
leal   7(%eax), %ecx
movl   %ecx, a
leal   14(%eax), %ecx
movl   %ecx, a
addl   $21, %eax
movl   %eax, a
ret
```

[3]

[3] compiled using LLVM - GCC 2.2 for IA32

Logic programming - mission critical

semantic preservation with CompCert



[4]

---

[4] http://compcert.inria.fr/