Logik in Clojure mit core.logic

Chris Weber und Julian Schmitt

11. Februar 2015

Inhaltsverzeichnis

1 Einführung

Diese Ausarbeitung wurde im Rahmen der Veranstaltung "Funktionale Programmierung mit Clojure" geschrieben. Sie beschäftigt sich speziell mit dem Clojure Paket core.logic und der logischen Programmierung. Das erste Kapitel beschreibt Grundlagen der Logischen Programmierung, um eine allgemeine Wissensbasis herzustellen. Im zweiten Kapitel untersuchen wir das Paket core.logic näher und beschreiben dessen Eigenheiten, Syntax und Funktionen. Nachfolgend im dritten Kapitel beschreiben wir zusätzliche Features die core.logic derzeit unterstützt und die den Funktionsumfang des Logik Paketes beträchtlich erhöhen. Zum Abschluss zeigen wir eine Anwendung von core.logic mit dem Einsteintest und versuchen diesen, mit den vorher gewonnen Informationen zu erklären.

2 Grundlegendes

In diesem Kapitel wird die logische Programmierung kurz vorgestellt, um die Grundlagen auf denen auch das Clojure Paket core.logic besteht vorwegzunehmen. Weiterhin sollen auch Grundzüge der relationalen Programmierung erklärt werden, auf der einige Funktionen der logischen Programmierung basieren.

2.1 Grundlagen der logische Programmierung

Logische Programmierung besteht nicht wie die funktionale Programmierung aus Folgen von Anweisungen, sondern aus Regeln und Fakten mit denen der Interpreter versucht Lösungsaussagen zu treffen. So gibt man zum Beispiel dem Interpreter die Regel, dass die Variable x eine Zahl sein soll, die gleich sein soll mit dem Ergebnis aus `2 + 3`.

Der Interpreter oder auch Lösungsmaschine oder Solver genannt, bekommt also ein Ziel (Goal) vorgegeben und versucht dieses mit Hilfe von Fakten Rückwärts zu lösen.

Ein logisches Programm besteht also aus einem oder mehreren Ausdrücken und einer Lösungsmaschine. Ein logischer Ausdruck ist ein Ziel, dass die Lösungsmaschine erreichen will.

Ein logischer Ausdruck besteht generell aus einer Menge an logischen Variablen und den entsprechenden Beschränkungen auf die Variablen. So stellt aus dem vorherigen Beispiel x die logische Variable dar und `x = 2 + 3` ist die Beschränkung auf x.

Die wichtigsten Funktionen die eine logische Programmiersprache ausmachen sind die Unifikation, die Einführung von logischen Variablen und die logische Disjunktion von Beschränkungen.

2.2 Relationen in der logischen Programmierung

Funktionen in der logischen Programmierung basieren oft auf Relationen, da diese es erlauben, Bedingungen rückwärts auszuwerten. Eine logische Programmiersprache wie zum Beispiel Prolog, die mit Fakten und Datenbanken arbeitet, erlaubt es in den Fakten, eigene Relationen darzustellen und diese hinterher mit den entsprechenden Funktionen auszuwerten. Darauf gehen wir näher im Kapitel *Weitere Features von core.logic* ein.

2.2.1 Beispiel

Eine Relation plus stellt eine Abbildung des Kreuzprodukts zweier natürlicher Zahlen auf eine natürliche Zahl dar.

```
plus: N x N -> N
    (a, b) |-> (+ a b)
```

a	b	plus
1	1	2
1	2	3
2	2	4

Somit können wir unseren Solver nutzen, um zu prüfen ob eine bestimmte Kombination von Argumenten erlaubt ist.

```
plus_o Relation  N x N x N
                -(1 1 1)- nicht erlaubt
                 (1 1 2) erlaubt
```

Relationale Programmierung kann rückwärts ausgewertet werden

```
(run* [q] (== q (plus_o (1 1 q))))
(run* [q] (== q (plus_o (q 1 3))))
(run* [q r] (== q (plus_o q r 3)))
```

3 core.logic

3.1 Allgemeines zu core.logic

Die Clojure-Erweiterung core.logic bietet die Möglichkeit, in Clojure, Prolog-ähnlich verschieden Programmierparadigmen zu verfolgen, wie z.B. die einfache logische Programmierung oder auch die logische Constraintprogrammierung. Damit soll eine einfache Möglichkeit geschaffen werden, bei der Lösung von logischen Problemen, die bestehenden Mittel (von core.logic) zu nutzen oder auch zu erweitern.

Momentan stehen David Nolen und Rich Hickey, Erfinder des Lisp-Dialekts Clojure hinter dem quelloffenen Projekt.

core.logic wird stetig weiterentwickelt und erhält immer wieder neue Features und Funktionen. Zum aktuellen Zeitpunkt schafft es core.logic mit der richtigen Implementierung ein Sudoku Rätsel innerhalb von wenigen Sekunden zu lösen. Leider ist core.logic nicht die effizienteste Implementierung eines logischen Programmierparadigmas und terminiert daher etwas langsamer als herkömmliche logische Programmiersprachen.

core.logic wurde auf Basis des Solvers miniKanren, der auf Scheme basiert, in die Clojure Welt eingeführt. Daher ist core.logic der miniKanren implementation sehr ähnlich und verwendet auch dessen Namenskonventionen und Funktionen.

Kanren ist Japanisch und kann in etwa mit Relation übersetzt werden. Da der Solver core.logic und auch miniKanren sehr stark von Relationen abhängig sind und diese nutzen um Bedingungen aufzulösen, ist es nicht überraschend, dass zumindest der Solver miniKanren danach benannt wurde.

Der Solver miniKanren lässt sich mit einem Satz in etwa so beschreiben: „Wenn miniKanren ein Ausdruck und eine gewünschte Ausgabe gegeben wird, kann es dies Rückwärts ausführen und findet dabei alle möglichen Eingaben zu dem Ausdruck die die gewünschte Ausgabe erzeugen."

3.1.1 Logische Ausdrücke

Ein logischer Ausdruck ist also eine Anweisung für den Solver und besteht aus den folgenden Teilen:

- eine Menge von logischen Variablen
- eine Menge von Beschränkungen auf die Werte, die die logischen Variablen annehmen können

3.1.2 Logische Variablen

Logische Variablen sind Container für einen nicht eindeutigen Wert. Das heißt, dass eine logische Variable mehrere Werte nacheinander annehmen kann, um diese auszugeben oder weiterzugeben. Logische Variablen können spezielle Werte haben, zum Beispiel _0. Dies soll darstellen, dass die entsprechende logische Variable jeden beliebigen Wert annehmen kann, um die Bedingungen zu erfüllen.

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

```
Ausgabe: [_0 _0]
```

Diese Ausgabe bedeutet, dass beide logischen Variablen q und r jeden beliebigen Wert annehmen können, um die Bedingungen zu erfüllen, dabei müssen sie aber beide den gleichen Wert annehmen.

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

```
Ausgabe: [_0 _1]
```

Bei dieser Anweisung können q und r auch jeden beliebigen Wert annehmen, dürfen dabei aber auch distinkt voneinander sein, um die Bedingungen zu erfüllen.

In core.logic gibt es zwei Wege, um logische Variablen einzuführen:

- `(run * [...] ...)`
- `(fresh [...] ...)`

Da `(run * [])` einen logischen Ausdruck einleitet, muss hier auch immer mindestens 1 logische Variable eingeführt werden. Weiterhin sind alle logischen Variablen immer nur in dem Bereich und allen tieferen Bereichen verfügbar in denen sie eingeführt wurden.
Beispiel:

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

Da die logische Variable x durch `fresh` eingeführt wurde, kann diese nur innerhalb des `fresh`-Bereichs genutzt werden. Außerhalb der Klammern von `fresh` ist x nicht mehr gültig. Die logische Variable q wurde allerdings von `run` eingeführt und ist daher auch innerhalb von `fresh` verfügbar und kann dort genutzt werden.

3.1.3 Beschränkungen

Beschränkungen oder auch Constraints sind Ausdrücke die die Werte die eine logische Variable annehmen kann, beschränken. Es können mehrere Beschränkungen existieren die untereinander in einer Konjunktion stehen:

```
(run* [q]
      (constraint -1)
      (constraint -2)
      (constraint -3)
)
```

Hier muss ein Wert alle 3 Constraints erfüllen, um als Wert von q angenommen werden zu können.

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

Im Beispiel muss ein Wert in den beiden Mengen `[1 2 3]` und `[2 3 4]` beinhaltet sein, um von q als Wert angenommen zu werden. Das Ergebnis wäre in diesem Beispiel: `[2 3]`.

3.2 Syntax

In diesem Kapitel soll die allgemeine Syntax von core.logic, die wichtigsten Funktionen und einige weiterführenden Funktionen vorgestellt werden. Weiterhin werden tiefergreifende Features vorgestellt und erklärt.

3.2.1 Allgemeine Syntax

Wie bereits in dem vorhergehenden Kapitel an einigen Beispielen zu sehen war, hat core.logic eine signifikante Syntax.

```
(run * [logic-variables] (logic-expressions in conjunction))
```

Dieser Ausdruck liest sich wie folgt: "Nimm die logischen Ausdrücke, lass den Solver diese lösen und gib alle Werte der logischen Variblen zurück die diese Ausdrücke erfüllen."

Um nicht bei jedem Aufruf der `run` Funktion alle Werte der logischen Variable zu bekommen, sondern nur endlich viele, kann man den * nach `run` durch eine Zahl ersetzen die der Anzahl der Werte entspricht die zurück gegeben werden sollen.

3.2.2 Die wichtigsten Funktionen

core.logic basiert, ähnlich wie miniKanren, auf 3 grundlegenden Funktionen.

`fresh`: Mit fresh lassen sich beliebig viele neue logische Variablen ins Programm einführen. Variablen die durch fresh eingeführt wurden, sind auch nur innerhalb von diesem gültig, d.h. lvars innerhalb von fresh müssen auf eine außerhalb von fresh gültige lvar übertragen werden.

`unify`: unify setzt lvars gleich. Entweder zu anderen lvars oder zu Werten. Mit unify lassen sich so zB lvars innerhalb von fresh auf eine lvar außerhalb von fresh übetragen.

`conde`: Mit conde (ähnlich zu cond aus dem clojure.core Paket) lassen sich Constraints so gesagt "verodern". Das heißt es erzeugt eine logische Disjunktion von Constraints.

Beispiel für `conde`:

```
(run* [q]
    (conde
        ((unify q 2))
          *OR*
        ((unify q 1) *AND* (unify q q))
          *OR*
        ((fresh [r s]
          (unify r 1)
            *AND*
          (unify s 2)
            *AND*
          (unify r q)
            *AND*
          (unify s q))
)
```

Das sind die 3 grundlegenden Funktionen von core.logic. Das gesamte Package beinhaltet aber natürlich noch viele mehr, Wie z.B. das eben gesehene (membero ...). Alle weiteren Funktionen im Package bauen aber auf den 3 Basis Funktionen auf. Höhere Funktionen folgen einer bestimmten Namenskonvention, zu sehen z.B. bei conde und memebero. Hiermit werden Funktionen in core.logic die schon im clojure.core existieren mit einem a, e, u oder o "verlängert", um diese von den regulären clojure Funktionen zu differenzieren und diese nicht zu überschreiben. Aber stehen diese Suffixe oft auch für ein bestimmtes Verhalten von Funktionen, sodass der Entiwckler auf den ersten Blick erkennen kann, wie diese Funktion in etwa arbeitet. Diese Namenskonventionen gibt es schon länger, und begründen ihre historische Entstehung in Sprachen wie Prolog und miniKanren.

- `conde`: Das e steht für "everyline" bzw. das jede Zeile von `conde` erfolgreich sein, bzw. true zurückgeben kann.

- `conda`: (Soft cut) Sobald der "HEAD" einer Bedingungsanweisung erfolgreich ist, liefert `conda true` zurück und ignoriert alle nachfolgenden Anweisungen. `conda` ist nicht-relational.
- `condu`: (Commited choice) Soblad der "HEAD" einer Bedingungsanweisung erfolgreich ist, werden die verbleibenden "goals" der Anweisung nur einmal ausgeführt. `condu` ist nicht-relational.
- `membero`, `anyo`: Das "o" bedeutet das hier eine Relation behandelt wird.

3.2.3 Aufruf des Solvers

Einfaches Beispiel:

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

`run 1`: Mit `run` wird der Solver gestartet und dieser soll das erste Ergebnis, das er bekommt, zurückgeben.

`[q]`: Das ist die logische Variable für die der Solver Werte suchen soll.

`(== 1 q)`: Das ist die Beschränkung auf die logische Variable. q wird hier mit 1 unifiziert und gibt damit vor, dass q = 1 sein muss damit diese Beschränkung erfüllt ist.

Werden mehrere Beschränkungen definiert, macht es für den Solver keinen Unterschied in welcher Reihenfolge diese stehen.

3.2.4 Höhere Funktionen

Neben den Funktionen `unify (==), fresh und conde` verfügt das Paket core.logic um einige weitere Funktionen die auf diesen 3 grundlegenden Funktionen aufbauen.

Dazu gehört zum Beispiel das bereits genannte `membero`:
`(membero x M)` beschränkt die logische Variable (in diesem Fall x) so, dass diese ein Element der Menge M sein muss, damit die Beschränkung erfüllt ist.

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

Dieses Beispiel würde die Ausgabe `(1, 2, 3)` zurückgeben, da q eine dieser 3 Zahlen annehmen kann, um ein Element der Menge `[1 2 3]` zu sein.

```
(defne membero
        [x l]
        ([_ [x . tail]])
        ([_ [head . tail]]
         (membero x tail)
        )
)
```

Die Definition von `membero` in core.logic. Diese besteht aus zwei Beschränkungen, `([_ [x . tail]])` und `([_ [head . tail]] (membero x tail))`. Während die erste Beschränkung erfüllt ist, wenn x das erste Element der Menge l ist, besagt die zweite, dass wenn x nicht das erste Element der Menge ist, dann ist x das erste Element der Menge tail (wobei tail die Menge l ohne deren erstes Element darstellt).

Weitere höhere Funktionen sind:

`(resto l r)`: `resto` schränkt die logische Variable so ein, dass r die Restmenge der Menge l ist. Das heißt r ist die Menge l ohne deren erstes Element.

Die Implementierung der Funktion `resto`:

```
(defn resto
        [l d]
        (fresh [a]
          (== (lcons a d) l))
)
```

`resto` werden zwei Variablen übergeben: l und d - wobei l die Gesamtmenge und d die Restmenge darstellt. Weiterhin wird aber noch der Kopf der Menge benötigt, welche mit `fresh [a]` eingeführt wird. Die Funktion `(lcons a d)` macht nichts anderes als aus den beiden Variablen a und d eine ordentliche Menge zu erstellen, mit a als Kopf und d als Restmenge. Diese soll dann der Menge l entsprechen und wird daher mit dieser unifiziert.

Beispiel zur Funktionsweise von `resto`:

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

q entspricht in diesem Beispiel der Menge (2 3 4).

`(conso x r s)`: `conso` schränkt die logische Variable so ein, dass x das erste Element einer Menge, r der Rest dieser Menge und s genau diese Menge ist.

```
(defn conso
        [a d l]
        (== (lcons a d) l)
)
```

`conso` und `resto` arbeiten auf die gleiche Art und Weise, nur, dass `conso` drei Variablen übergeben bekommt (Kopf, Rest- und Gesamtmenge). `(lcons a d)` erstellt wieder aus Kopf und Restmenge eine komplette Menge, die dann mit der übergebenen Gesamtmenge unifiziert wird.

Beispiele zur Funktionsweise von `conso`:

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

q entspricht hier der Menge (1 2 3 4)

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

q entspricht hier 1

4 Weitere Features von core.logic

4.1 Datenbanken

Bei der funktionalen Programmierung und/oder auch der Relationalen, bietet es sich an, über eine Faktenbasis und deren verschiedenen Relationen eine Art Datenbank zu erstellen um dort Informationen abfragen zu können. Mit core.logic ist so etwas relativ schnell und komfortabel mit dem Unterpacket von core.logic, core.logic.pldb möglich.

4.1.1 Model

Unser Model ist eine gewöhnliche italienische Familie. Dort exisiteren Väter, Mütter und Kinder. Ein Vater hat ein Kind und eine Mutter hat ein Kind. Das sind dann auch unsere Relationen.

```
Relation V : X ist Vater von Y

Relation M : X ist Mutter von Y
```

4.1.2 Code

Die Packete `core.logic` und hauptsächlich `core.logic.pldb` mit folgendem Befehl bekannt machen, dabei aber das `==` vom clojure.core nicht berücksichtigen.

```
(ns logic.core
        (:refer-clojure :exclude [==])
        (:use [clojure.core.logic])
        (:use [clojure.core.logic.pldb]))
```

Definieren der oben formulierten Relationen im Programm. Hierbei werden diese an ein Symbol und damit an ein Objekt im Speicher gebunden.

```
(db-rel father Father Child)
(db-rel mother Mother Child)
```

Die oben programmintern gebundenen Relationen können jetzt verwendet werden um mit `db` eine Repräsentation im Speicher zu erzeugen und diese wieder an Symbole zu binden. Dabei bekommt `db` eine Anzahl von Vektoren, die am Anfang ein Relations-Objekt ("Instanz" von db-rel) stehen haben und danach Argumente von der Größe des Tupel der Relation folgen um die Relation und des Tupel darzustellen.

```
(def dbf (db [father 'Vito 'Michael]
             [father 'Vito 'Sonny]
             [father 'Vito 'Fredo]
             [father 'Michael 'Anthony]
             [father 'Michael 'Mary]
             [father 'Sonny 'Vicent]
             [father 'Sonny 'Francesca]
             [father 'Sonny 'Kathryn]
             [father 'Sonny 'Frank]
             [father 'Sonny 'Santino]))

(def dbm (db [mother 'Carmela 'Michael]
             [mother 'Carmela 'Sonny]
             [mother 'Carmela 'Fredo]
             [mother 'Kay 'Mary]
             [mother 'Kay 'Anthony]
             [mother 'Sandra 'Francesca]
             [mother 'Sandra 'Kathryn]
             [mother 'Sandra 'Frank]
             [mother 'Sandra 'Santino]))
```

Der Zugriff auf die Tabellen unseres realtionalen Datenbankmodels erfolgt über `with-db` auf eine Datenbank und über `with-dbs` auf einen Vektor von Datenbanken. Der Aufruf des Solvers erfolgt nach altbekanntem Muster. Die Benutzung von codeconde bietet sich in diesem Fall an, da wir so mehrere Filter verodern können. In diesem Fall entspricht das einem SQL-ähnlichen `JOIN` auf das Kind q.

```
(defn children [p]
  "Returns a list of children of p"
  (with-dbs [dbf dbm]
        (run* [q]
                (conde ((father p q)) ((mother p q))))))
```

Somit bekommen wir mit folgendem Code folgende Ausgabe:

```
(children 'Vito)
; => (Michael Sonny Fredo)
```

Dabei funktioniert nur das Symbol `'Vito` und nicht der String `"Vito"`

```
(children "Vito")
; => ()
```

Ein ähnliches Vorgehen für die Ermittlung der Eltern eines Kindes "c". Was vielleicht verwirrt, das wir bei dem Suffix der Funktion relativ willkürlich ein e gewählt haben. Siehe Kapitel ??? von Syntax.

```
(defn parentse [c]
  "Returns a list of parents of c"
  (with-dbs [dbm dbf]
        (run* [q]
                (conde ((father q c)) ((mother q c))))))
```

Wird nichts gefunden bekommen wir eine Leere Menge.

```
(parentse 'Vito)
; => ()

(parentse 'Anthony)
; => (Michael Kay)
```

Das Ermitteln des Vaters bzw. der Mutter ist sehr ähnlich. Und nur deshalb nur der Vollständigkeit halber erwähnenswert.

```
(defn fathero [c]
  "Returns the father of c"
  (with-db dbf
        (run* [q]
                (father q c))))

(defn mothero [c]
  "Returns the mother of c"
  (with-db dbm (run* [q] (mother  q c))))
```

Bei der Ermittlung der Großenkel eines Großvaters wird es schon etwas komplizierter. Wir möchten alle Enkel in der Abfrage erwischen, das sind anders gesagt, die Kinder der Kinder. Genauer: Wir legen uns per `fresh` zwei neue logische Variablen `x` und `y` an. Unser gegebenes `g` ist der Großvater und damit der Anfang unserer "verschlungenen" Relationenkette. Jedes q aus der Menge aller Kinder der `dbf` Tabelle, welches einen Vater `x` hat, welcher `g` seinen Vater nennt, wird in die Lösungsmenge aufgenommen. In PostgreSQL wäre so etwas mit `RECURSIVE` möglich oder auch mit einem `JOIN` auf einer Tabelle.

```
(defn grandchild [g]
  "Returns the grandchild(s) of g"
  (with-dbs [dbf] (run* [q] (fresh [x y] (father g x) (father x y) (== q y)))))
```

Damit bekommen wir mit der Eingabe `'Vito`, alle Enkel von ihm.

```
(grandchild 'Vito)
; (Francesca Santino Frank Mary Kathryn Vicent Anthony)
```

4.2 Constraint Logic Programming

Nach dem aktuellen Stand unterstützt core.logic bereits die beiden Formen des Constraint Logic Programming "disequality constraints over trees" auch CLP(Tree) und "constraints over finite domains" auch CLP(FD). Constraint logic programming ist sehr ähnlich zur regulären logischen Programmierung, allerdings arbeitet der Interpreter mit Backtracking und einem "constraint store" in den er alle Constraints schreiben kann, um die Erfüllbarkeit aller Constraints auf einer logischen Variable einfacher überprüfen zu können.

4.2.1 CLP(Tree)

Diese Form des constraint logic programming emuliert die reguläre logische Programmierung, in dem der Interpreter nur Subsitutionen als constraints in den "constraint store" schreibt. Die Terme der Substitutionen sind Variablen, Konstanten und Funktionen die auf andere Terme angewandt werden. Die einzigen Constraints die als solche anerkannt werden sind Gleichsetzungen (==) und Ungleichsetzungen (!=). core.logic führt dazu erstmals einen neuen Operator ein: != . Dieser Garantiert, dass zwei gegebene Terme niemals unifiziert werden können. Das einfachste Beispiel dazu ist eine Überprüfung, ob eine Variable nicht gleich einem Wert ist:

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

Das Ergebnis dazu sieht so aus: `((_0 :- (!= _0 1)))` Das liest sich wie folgt: q kann jeden Wert annehmen, solange dieser Wert nicht gleich 1 ist.

Die Anwendung von Ungleichheiten wird bei etwas komplexeren Termen auch wesentlich interessanter:

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

Das Ergebnis zu diesem Beispiel ist interessant, da sich der Ausdruck wie folgt beschreiben lässt: Es sollte niemals vorkommen, dass x = 2 ist **und** dass y = 1 ist. Das bedeutet, ist y = 5 ist die Bedingung erfolgreich und wird abgebrochen, ist aber y = 1 wird weiter überprüft, ob x jemals 2 wird.

4.2.2 CLP(FD)

Constraint logic programming over finite domains arbeitet mit endlichen Mengen und weist diese Variablen zu. So kann eine Variable zum Beispiel eine Zahl zwischen 1 und 5 annehmen. Das Paket core.logic.fd führt auch noch einige weitere neue Operatoren ein:

- `+`
- `-`
- `*`
- `quot`
- `==`
- `!=`
- `<`
- `<=`

- `>`
- `>=`
- `distinct`

Logische Variablen werden hier mit `fd/in` eingeführt.

```
(run * [q]
       (fd/in q (fd/interval 1 5))
)
```

q erhält hier die Werte (1 2 3 4 5) Ein weitere Beispiel dazu:

```
(run * [q]
       (fresh [x y]
       (fd/in x y (fd/interval 1 10))
       (fd/+ x y 10)
       (== q [x y]))
)
```

Ergebnis: `([1 9][2 8][3 7][4 6][5 5][6 4][7 3][8 2][9 1])`

Weiterhin gibt es noch den sehr nützlichen Operator `distinct`. Dieser garantiert, dass alle ihm übergebenen Variablen die auf finite domains aufbauen niemals den gleichen Wert annehmen:

```
(run * [q]
       (fresh [x y]
       (fd/in x y (fd/interval 1 10))
       (fd/+ x y 10)
       (fd/distinct [x y])
       (== q [x y]))
)
```

Das Ergebnis hier: `([1 9] [2 8] [3 7] [4 6] [6 4] [7 3] [8 2] [9 1])`
Hierbei fällt auf: `[5 5]` ist nicht mehr in der Ergebnismenge enthalten, da dabei x und y nicht unterschiedlich sind.

5 Einstein-Test oder Zebrapuzzle

Bei diesem Rätsel geht es darum, aus einer Menge von 5 Personen, die sich alle jeweils durch die Farbe ihres Hauses, ihr Getränk, ihr Haustier, ihre Zigarettenmarke und ihre Nationalität unterscheiden, mithilfe von gegebenen Informationen und einem logischen Lösungsansatz, genau eine Person mit einer gewissen Eigenschaft herauszufinden. Näheres hierzu z.B. auf Wikipedia.

5.1 Code

Das entsprechendes Codebeispiel kann auf folgender Seite `https://github.com/swannodette/logic-tutorial#zebras` gefunden werden. Nachfolgend werden die im Codebeispiel definierten Methoden erklärt.

5.1.1 righto

```
(defne righto [x y l]
  ([_ _ [x y . ?r]])
  ([_ _ [_ . ?r]] (righto x y ?r)))
```

Diese Methode erzeugt alle Beschränkungen, die wir benötigen, damit "y" rechts von "x" steht. Ein rekursiver Aufruf sorgt stößt den Prozess sooft wieder an bis der Rest, dargstellt durch das `?r` behandelt wurde. Genauer werden also die Constraints zurückgegeben, oder auch Goals, die der Solver benötigt um unser Ergebnis zu errechnen.

5.1.2 nexto

```
(defn nexto [x y l]
  (conde
    ((righto x y l))
    ((righto y x l))))
```

Diese Methode erzeugt alle Beschränkungen, die wir benötigen, damit "y" links oder rechts neben "x" steht. Dazu wird die oben erklärte Methode `righto` verwendet.

5.1.3 zebrao

```
(defn zebrao [hs]
  (macro/symbol-macrolet [_ (lvar)]
    (all
     (== [_ _ [_ _ 'milk _ _] _ _] hs)
     (firsto hs ['norwegian _ _ _ _])
     (nexto ['norwegian _ _ _ _] [_ _ _ _ 'blue] hs)
     (righto [_ _ _ _ 'ivory] [_ _ _ _ 'green] hs)
     (membero ['englishman _ _ _ 'red] hs)
     (membero [_ 'kools _ _ 'yellow] hs)
     (membero ['spaniard _ _ 'dog _] hs)
     (membero [_ _ 'coffee _ 'green] hs)
     (membero ['ukrainian _ 'tea _ _] hs)
     (membero [_ 'lucky-strikes 'oj _ _] hs)
     (membero ['japanese 'parliaments _ _ _] hs)
     (membero [_ 'oldgolds _ 'snails _] hs)
     (nexto [_ _ _ 'horse _] [_ 'kools _ _ _] hs)
     (nexto [_ _ _ 'fox _] [_ 'chesterfields _ _ _] hs))))
```

Diese Methode enthält sämtliche Regeln des Puzzles. Durch die Wahl von sprechenden Namen, der verwendeten und selbst definierten Methoden, sind die Regeln sehr gut abzulesen. Die erste Zeile enthält z.B. zwei Regeln. Einmal die Regel, das es fünf Häuser gibt und die Regel, das die Person im mittleren Haus trinkt Milch trinkt. Die zweite Regel sagt aus, das die Person ganz links (firsto, "der Erste") norwegisch ist. Die Dritte, das neben der norwegischen Person ein blaues Haus steht und so weiter. Im Programmcode werden in der zweiten Zeile die Zeichen "lvar" an das Symbol "_" gebunden. Das erspart einige Zeichen Code und erhöht die Lesbarkeit. Die fünf Häuser mit jeweils einer Person und deren fünf verschiedene Eigenschaften, werden intern durch eine 5x5 Matrix dargestellt. Ein Vektor der größe fünf für die Darstellung der Häuser, und jeweils für jedes Haus bzw. die Person und deren Eigenschaften ein Vektor der Größe fünf. Ein "Haus-Vektor" hat dabei folgende Bedeutung:

```
['Nationalität' 'Zigarettenmarke' 'Getränk' 'Haustier' 'Hausfarbe']
```

Somit existiert hier eine Relation "Person" und "Nachbarschaft". Diese wären wie folgt definiert:

```
Relation P : Ist A, hat  Keine ahnung, Spaeter
Relation N : P wohnt neben Q; Tupel: (P, Q)
```

5.2 Lösung des Rätsels

Nach der Ausführung, und damit der Definition der obigen Codezeilen, führt man folgenden Code aus.

```
(run 1 [q] (zebrao q))
```

Damit führen wir unsere Lösungsmaschine, mit den durch `zebrao` erstellten Constraints aus und lassen uns durch die Angabe des Arguments "1" die Erste Lösung zurückgeben.

Ausgabe der Lösung:

```
(([norwegian kools _.0 fox yellow]
  [ukrainian chesterfields tea horse blue]
  [englishman oldgolds milk snails red]
  [spaniard lucky-strikes oj dog ivory]
  [japanese parliaments coffee _.1 green]])
```

In diesem Fall kann man die Ausgangsfragen beantworten:

1. Wer trinkt Wasser?
2. Wer hat den Fisch?

Die Ausgabe sagt aus, dass wir in jedem Fall an beiden Stellen zwei voneinander verschiedene Ergebnisse haben. Somit wäre eine plausible Lösung in unserem Fall, das der Norweger Wasser trinkt und der Japaner einen Fisch hat. Ob Fisch und Wasser oder andere Haustiere und Getränke gefragt sind, kommt natürlich auf die Fragestellung an.