

Logik in Clojure mit core.logic

Chris Weber und Julian Schmitt

10. Februar 2015

Inhaltsverzeichnis

1	Einführung	3
1.1	Grundlagen der logische Programmierung	3
1.2	Relationale Programmierung	3
2	Grundsätze	3
2.1	core.logic	3
2.1.1	Logische Ausdrücke	4
2.2	Logische Variablen	4
2.3	Beschränkungen	4
3	Syntax	4
3.1	Allgemeines	4
3.2	Wichtige Funktionen	4
3.3	Aufruf der Instanz eines Lösungsautomaten oder auch Solver	5
4	Einstein-Test oder Zebrapuzzle	5
4.1	Code	5
4.1.1	righto	6
4.1.2	nexto	6
4.1.3	zebrao	6

1 Einführung

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.

1.1 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.

1.2 Relationale Programmierung

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

<code>plus : N x N -> N</code>			
<code>(a, b) -> (+ a b)</code>	<code>a</code>	<code>b</code>	<code>plus</code>
	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)))
```

2 Grundsätze

2.1 core.logic

`core.logic` ist eine Implementierung des auf Scheme basierenden Solvers: `miniKanren`

Kanren ist Japanisch und bedeutet so viel wie Relation.

`miniKanren` in einem Satz: Wenn `miniKanren` ein Ausdruck und eine gewünschte Ausgabe gegeben wird, kann es dies "Rückwärts" ausführen und findet dabei alle möglichen Eingaben zu dem Ausdruck der die gewünschte Ausgabe erzeugt hat.

2.1.1 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

Je nach Ausdruck, ist die Anzahl der logischen Variablen, die diese Einführen unterschiedlich. (run ..) zum Beispiel, kann nur eine lvar einführen.

2.2 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

Tafelbild: (run* [q] (== q q)) ; Rückgabe -j _0 In diesem Fall wird _0 zurückgegeben, was so viel bedeutet wie: q kann jeden beliebigen Wert annehmen und erfüllt immer die Beschränkung. Für den Fall, dass wir mehrere logische Variablen haben (z.B. 2) und beide können jeden beliebigen Wert annehmen, müssen aber gleich sein ist die Rückgabe (_0 _0), können beide distinkt voneinander sein: (_0 _1). Logische Variablen werden auf 2 Arten in ein logisches Programm eingeführt: (run* [q] ... (fresh [...] ...)) Während (run* ...) immer nur 1 logische Variable einführen kann, führt (fresh ...) beliebig viele ein, muss aber innerhalb von einem (run* ...) Befehl stehen.

2.3 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. Ergebnis wäre hier [2 3] .

3 Syntax

3.1 Allgemeines

3.2 Wichtige 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 übertragen.

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:

```
(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, wie z.B. zu sehen bei conde und memebro, werden Funktionen in core.logic die schon im clojure.core existieren mit einem a,e,u oder o um diese von den regulären clojure Funktionen zu differenzieren und diese nicht zu überschreiben.

Ein nachgestelltes

- a steht für
- e steht für
- u steht für
- o steht für die Rückgabe von Goals bzw. einer Relation

3.3 Aufruf der Instanz eines Lösungsautomaten oder auch Solver

```
(run* [logic-variable] &constraints)
```

Tafelbild: (run* [q] (== 1 q)) ; Rückgabe -1 (run* ...) -1 Befehl für den Solver; * ist Anzahl der Ergebnisse, kann auch entsprechend eine Zahl sein. [q] -1 ist die logische Variable $f\tilde{A}_{\frac{1}{4}}r$ die ein oder mehrere Werte gesucht wird (== 1 q) -1 ist die Beschränkung $f\tilde{A}_{\frac{1}{4}}r$ q. "Gib alle Werte $f\tilde{A}_{\frac{1}{4}}r$ q zurück $f\tilde{A}_{\frac{1}{4}}r$ die gilt: q == 1"

4 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 z.B. auf Wikipedia.

4.1 Code

Das entsprechende Codebeispiel kann auf folgender Seite <https://github.com/swannodette/logic-tutorial#zebras> gefunden werden.

Nachfolgend werden die im Codebeispiel definierten Methoden erklärt.

4.1.1 righto

```
(defne righto [x y 1]
  ([_ _ [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 dafür, dass der Prozess so oft wieder an bis der Rest, dargestellt durch das ?r behandelt wurde. Genauer werden also die Constraints zurückgegeben, oder auch Goals, die der Solver benötigt um unser Ergebnis zu errechnen.

4.1.2 nexto

```
(defn nexto [x y 1]
  (conde
    ((righto x y 1))
    ((righto y x 1))))
```

Diese Methode erzeugt Prädikate um alle Permutationen eines Objektes zu bekommen, wenn es sich daneben befindet. Auch hier reden wir bei der Rückgabe wieder von Constraints/Goals.

4.1.3 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))))
```

Die fünf Häuser mit jeweils einer Person und deren fünf verschiedene Eigenschaften, werden intern durch eine Matrix dargestellt. Ein Vektor der Größe fünf für die Darstellung der Häuser, und jeweils für jedes Haus ein Vektor der Größe fünf. Ein "Haus-Vektor" hat dabei folgende Bedeutung:

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

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, dass es fünf Häuser gibt und die Regel, dass die Person im mittleren Haus trinkt Milch trinkt. Die zweite Regel sagt aus, dass die Person ganz links (firsto, "der Erste") norwegisch ist. Die Dritte, dass 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.