Logik in Clojure mit core.logic

Chris Weber und Julian Schmitt

10. Februar 2015

Inhaltsverzeichnis

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.

2 Relationale Programmierung

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

Die Clojure-Erweiterung core.logic bietet die Möglichkeit, in Clojure, Prolog-ähnlich verschieden Programmierparadigmen zu verfolgen, wie z.B. Relationale Programmierung [...] oder auch 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 (s. Wikipedia) hinter dem quelloffenen Projekt.

3.2

4 Grundsätze

4.1 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.

4.1.1 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

4.2 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.

4.3 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]`.

5 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.

5.1 Allgemeines

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.

5.2 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 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, wie z.B. zu sehen bei conde und memebero, 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

5.3 Aufruf der Instanz eines Lösungsautomaten oder auch Solver

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.

5.4 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.

```
(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.

```
(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

6 Einstein-Test oder Zebrapuzzle

6.1 Hintergrund

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.

6.2 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.

6.2.1 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.

6.2.2 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.

6.2.3 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']
```

6.3 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, das 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 (Kommt auf die Fragestellung an).