Noch ein Bier

Clojure – Teil 3: Nebenläufigkeit

Burkhard Neppert, Stefan Tilkov

Clojure kombiniert das Beste aus Lisp mit einer perfekten Java-Integration. In den ersten beiden Teilen dieser Clojure-Einführung haben Sie die Syntax, Funktionen höherer Ordnung, Datenstrukturen und die Integration mit Java kennengelernt. Dieser abschließende Teil behandelt die Programmierung paralleler Anwendungen an Beispielen wie der Simulation einer gut besuchten Bar und ums Besteck konkurrierenden Gästen.

Veränderbare Zustände und parallele Programme

In vielen Programmiersprachen sind Variablen veränderbare Speicherstellen, denen man Werte zuweisen kann. Diese Art von Variablen ist problematisch, wenn verschiedene Threads schreibend auf eine darauf zugreifen: In diesem Fall muss der Programmierer den Zugriff (schreibend und lesend) auf diese Speicherstellen explizit im Programmcode koordinieren, z. B. durch Javas `synchronized`-Blöcke. Die Erfahrung hat gezeigt, dass diese Art der Programmierung schwierig und überdies auch schlecht zu testen ist.

Ein zentrales Entwurfsziel für Clojure ist, eine gute Unterstützung für die Erstellung nebenläufiger Programme zu bieten. Um dies zu erreichen, werden in Clojure Instanzen von Datentypen als unveränderliche („immutable“) Einheiten betrachtet und verschiedene Sprachmittel implementiert, mit denen die Zuordnung von Werten zu Speicherstellen kontrolliert geändert werden kann. Dazu zählen Vars, Agenten, Referenzen und Transaktionen sowie Atome.

Persistent Collections und unveränderliche Werte

Ein Mittel zur Vereinfachung parallelen Codes ist es, Datentypen als unveränderliche Werte zu implementieren. Bei der kurzen Einführung der strukturierten Datentypen, der sogenannten Collections, im letzten Teil ist Ihnen vielleicht aufgefallen, dass keine Funktionen erwähnt wurden, die eine Collection ändern. Stattdessen hieß es stets sinngemäß „... liefert eine neue Liste/Vektor ... als Rückgabewert“. Clojures Collection-Instanzen sind unveränderlich, da bei dieser Art der Programmierung keine Notwendigkeit besteht, parallelen Zugriff auf Instanzen mit Locks zu koordinieren: Es muss kein schreibender Zugriff serialisiert werden und es besteht auch keine Gefahr, dass ein inkonsistenter Zustand gelesen wird.

Eine naive Implementierung von Collections mit Wert-Semantik würde eine Kopie der Elemente in der Collection erzeugen, mit den entsprechenden negativen Konsequenzen auf die Laufzeit und den Speicherplatzbedarf. Um diese negativen Effekte auf ein praktikables Maß zu reduzieren, verwendet Clojure spezielle Implementierungen für seine Collection-Klassen. Die Idee ist dabei, dass mehrere Collections gleiche Teile gemeinsam nutzen („structural sharing“). Referenzen auf die Werte sind dabei von den Werten getrennt.

Am einfachsten kann man diesen Ansatz anhand verketteter Listen zeigen. Der Grundbaustein einer verketteten Liste ist eine „Zelle“ mit zwei Feldern. Das erste Feld enthält einen Verweis auf das Listenelement, das zweite einen auf die nachfolgende Zelle der Liste (s. Abb. 1). Das Einfügen eines Elements am Beginn kann in konstanter Zeit geschehen; die Referenzen zeigen dabei weiterhin auf das Element, das für sie den Beginn der Liste repräsentiert. Den Clojure-Datenstrukturen wie Listen, Maps oder Vektoren liegen komplexere Datenstrukturen zugrunde, aber das Prinzip ist das gleiche (s. [Hick09]).

Clojures unveränderliche Collection-Klassen weisen die gleiche Laufzeitkomplexität für die charakteristischen Operationen auf, wie die veränderlichen Implementierungen; mit Ausnahme von Vektor, hier ist die Zugriffszeit nicht O(1), sondern O(log_32(n)).

In den allermeisten Fällen ist ein Programmierstil, der ohne Seiteneffekte auskommt, dem mit Seiteneffekten vorzuziehen. Gelegentlich ist die Veränderung eines Wertes einer Variablen aber doch die einfachste Art der Implementierung. Für diese Fälle besitzt Clojure Sprachelemente, mit denen Seiteneffekte in parallelen Programmen kontrolliert werden können.

Atome

Ein Atom koordiniert den parallelen synchronen Zugriff auf eine Variable aus verschiedenen Threads. Wie der Name bereits andeutet, ist der Zugriff atomar, d. h. es ist garantiert, dass der Wert entweder vollständig oder gar nicht geändert wird und dass keine inkonsistenten Zustände von anderen Threads gelesen werden können.

Ein neues „Atom“ wird mit der Funktion `atom` erzeugt, der Wert mit der Funktion `(deref <atom>)` oder mit `@<atom>` abgefragt:

```
>> (def an-atom (atom <anfangswert>))
>> @an-atom ;; => <anfangswert>
```

Der Wert eines Atoms wird mit `swap!` verändert. Diese Funktion erhält das Atom, dessen Wert geändert werden soll, als erstes Argument und als zweites Argument eine Funktion `f`, die aus dem aktuellen Wert des Atoms einen neuen Wert berechnet. Optional kann `f` noch weitere Argumente erhalten, die beim Aufruf von `swap!` nach `f` anzugeben sind. Der Rückgabewert von `swap!` ist der neue Wert des Atoms:

```
(swap! ein-atom (fn [wert-von-atom] …))
```

Als einfaches Beispiel soll ein thread-sicherer Zähler dienen. Die Funktion `neuer-zaehler` erzeugt Zähler-Funktionen, die in ihrem lokalen Bindungskontext (`let […]`) auf ein Atom namens `zaehler` zugreifen können und dessen Wert erhöhen:

```
>> (defn neuer-zaehler []
     (let [zaehler (atom 0)
          addiere (fn [zaehlerstand x] (+ zaehlerstand x))]
       (fn ([]  (swap! zaehler addiere 1))
           ([n] (swap! zaehler addiere n)))))

>> (def ein-zaehler (neuer-zaehler))
```

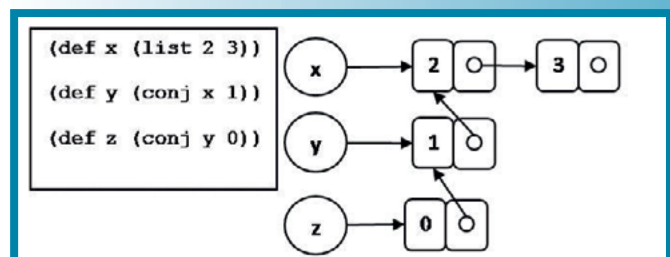

Abb. 1: Wiederverwendung von Listenelementen

```
>> (ein-zaehler)     ==> 1
>> (ein-zaehler 10) ==> 11
```

Wichtig ist, dass die Funktion, die den neuen Wert des Atoms berechnet, mehrfach aufgerufen werden kann, wenn mehrere Threads zeitgleich schreiben. Sie sollte daher keine Seiteneffekte haben.

Agenten

Eine weitere Möglichkeit, mit der in Clojure veränderliche Zustände realisiert werden können, sind sogenannte Agenten. Ein Agent besitzt einen eigenen Zustand, der jederzeit von anderen Threads mit `(deref <agent>)` bzw. `@<agent>` gelesen werden, aber nur durch den Agenten selbst verändert werden kann. Ein Agent wird durch die Funktion `(agent <anfangszustand>)` erzeugt.

Die Zustandsänderung eines Agenten wird durch die Funktion `(send <agent> <f>)` angestoßen. Diese Funktion erhält als erstes Argument den Agenten, dessen Zustand verändert werden soll, und als zweites Argument eine Funktion `f`, die den neuen Agentenzustand berechnet, optional gefolgt von weiteren Argumenten für `f`. Während die Änderung bei Atomen synchron geschieht, erfolgt die Ausführung von `f` asynchron in einem eigenen Thread aus einem Threadpool des Java-Laufzeitsystems. Bei der Auswertung wird `f` auf den aktuellen Zustand des Agenten und die optionalen Argumenten von `send` angewendet und das Ergebnis als neuer Agentenzustand gesetzt. Der sendende Thread erhält stets den Agenten als Ergebnis des `send`-Aufrufs.

Für Fälle, in denen die Berechnung des Agentenzustandes längere Zeit laufen oder blockieren kann (z. B. durch I/O), besitzt Clojure die Funktion `send-off`. Sie arbeitet im Prinzip wie `send`, nutzt jedoch keinen beschränkten Threadpool, sondern erzeugt bei Bedarf neue Threads.

Kleines Agententreffen

Der Umgang mit Agenten soll durch ein Beispiel erklärt werden, das Personen an einer Bar simulieren soll. Unsere Agenten repräsentieren Gäste, deren Zustand durch den Namen und die Getränkeaufnahmekapazität beschrieben wird. Die Gäste bestellen solange Getränke bei einem Bartender und trinken sie sofort, bis sie genug getrunken haben oder dem Bartender die Getränke ausgehen.

Die Gäste beschreiben wir mit einer aus Teil 2 [NeTi10b] bekannten `StructMap`, die den Namen des Gastes und seine verbleibende Getränkeaufnahmekapazität enthält:

```
>> (defstruct bar-gast :anrede :kapazitaet)
```

Durch

```
>> (def gaeste (map agent
                 [(struct bar-gast :Mr_Bond 7)
                  (struct bar-gast :Mr_Leitner 5)
                  (struct bar-gast :Q 2)
                  (struct bar-gast :M 3)]))
```

erzeugen wir die Agenten der Gäste mit ihrer noch verbliebenen Kapazität. Wie bereits weiter oben erwähnt, liefert die Funktion `(map <f> <coll>)` eine Liste zurück, die durch Anwendung der Funktion `f` auf jedes Element der Liste `coll` entsteht. In diesem Fall wird also die Funktion `agent` nacheinander für die Elemente des Vektors aufgerufen.

Als weiteren Agenten benötigen wir noch einen Bartender, dessen Zustand den verfügbaren Getränkevorrat beschreibt:

```
>> (def bartender (agent 10))
```

Die Funktion `bestellen` soll einen Bestellvorgang beim Bartender simulieren. Ihr erstes Argument ist der zum Zeitpunkt des Aufrufs vorhandene Getränkevorrat, das zweite Argument der Agent des Gastes, der die Bestellung aufgegeben hat. Sind noch Getränke vorhanden, wird an den Agenten des Gastes die `cheers`-Funktion gesendet und der Getränkevorrat verringert, andernfalls nur eine Nachricht ausgegeben:

```
>> (defn cheers [_] :dummy_wg_zyklischer_abhängigkeit)
>> (defn bestellen [vorrat gast]
     (if (> vorrat 0)
       (do
         (println "Cheers to you" (@gast :anrede))
         (send gast cheers) ;; Rückmeldung an bestellenden Agenten
         (- vorrat 1))
       (do
         (println "Sorry," (@gast :anrede))
         0)))
```

Die `cheers`-Funktion realisiert das Verhalten eines Gastes, wenn er ein Getränk bekommt: Sie erhält als Argument den Zustand des Gastes und liefert als Ergebnis einen neuen Zustand mit um 1 verringerter Kapazität, der Gast trinkt in jedem Fall. Wenn die Kapazität nach dem Trinken noch größer 0 (also die aktuelle > 1) ist, wird noch eine weitere Bestellung an den Bartender-Agenten gesendet. Durch `*agent*` kann auf den Agenten zugegriffen werden, in dessen Thread die Funktion aufgerufen wird, und so dem Bartender-Agenten mitgeteilt werden, welcher Agent das Getränk bekommen soll:

```
>> (defn cheers [gast]
     (if (> (gast :kapazitaet) 1)
       (send bartender bestellen *agent*))

     (assoc  gast :kapazitaet (- (gast :kapazitaet) 1)))
```

Obwohl im Code von `cheers` das Senden der Bestellung vor der Verringerung der Kapazität steht, sorgt das Clojure-Laufzeitsystem dafür, dass Nachrichten an andere Agenten erst dann versendet werden, wenn der Zustand des Agenten aktualisiert wurde. Dieses Verhalten kann mit der Funktion `release-pending-sends` geändert werden, alle bisherigen `send`-Aufrufe im Agenten werden sofort ausgeführt.

Die kleine Bar-Simulation wird gestartet, indem alle Gäste eine Bestellung an den Bartender-Agenten senden:

```
>> (dorun (map (fn [a] (send bartender bestellen a)) gaeste))
```

Da `map` eine „lazy sequence“ erzeugt, in der nur das erste Element berechnet wird, wir in unserem Beispiel aber an den Seiteneffekten von `place-order` interessiert sind, muss die Auswertung aller Listenelemente mit `dorun` erzwungen werden. Die Funktion erzwingt wie das oben erwähnt `doall` die Auswertung einer „lazy sequence“, baut jedoch keine Liste mit den Ergebnissen auf.

Auf Agenten warten

Die Verarbeitung eines Agentenaufrufs erfolgt asynchron in einem eigenen Thread des Agenten, der aufrufende Thread arbeitet sofort weiter. Mit der Funktion `await` kann der aufrufende Thread solange angehalten werden, bis sämtliche von ihm angestoßenen Agenten-Aktionen beendet wurden. Der folgende Code

```
>> (do
     (dorun (map (fn [a] (send bartender bestellen a)) gaeste))
     (apply await gaeste)
     (println „Agenten: „
           (map (fn [g] (vector(@g :anrede) (@g :kapazitaet)))
                gaeste)))
```

startet alle Agenten des letzten Beispiels, wartet bis alle Agenten die `send`-Aktion ausgeführt haben, und gibt dann den Zustand der Agenten aus. Sie werden bemerken, dass nach `await`

und der Ausgabe von **„Agenten ….** „ offensichtlich noch weitere Nachrichten an die Agenten gesendet werden. Ursache ist, dass diese „sends“ aus den Threads der Agenten heraus aufgerufen werden und nicht aus dem Thread, der **await** aufruft.

Auch Agenten machen Fehler

Treten in einer Aktion eines Agenten Exceptions auf, werden diese im Agenten gespeichert und können mit **agent-errors** abgefragt werden:

```
>> (def zahl (agent 0))
>> (agent-errors (send zahl #(/ 1 %)))
;; ==> (#<ArithmeticException: Divide by zero>)
```

Sendet ein Thread Nachrichten an einen Agenten, der sich im „Fehler“-Zustand befindet, resultiert das in einer RuntimeException im sendenden Thread. Der Agent kann durch **clear-agent-errors** wieder in einen arbeitsfähigen Zustand versetzt werden:

```
>> (send zahl #(+ 1 %))
;; ==> RuntimeException: Agent is failed, needs restart
>> (clear-agent-errors zahl) ;; ==> 0
>> (send zahl #(+ 1 %))
```

Software Transactional Memory

Agenten sind ein geeignetes Mittel, nebenläufige Veränderungen an einem Zustand zu synchronisieren. Eine gemeinsame Veränderung mehrerer Zustände ist mit Agenten nur schwer zu realisieren. Für diesen Anwendungsfall ist in Clojure mit „Software Transactional Memory“ (STM) ein Mechanismus realisiert, der ähnliche Eigenschaften wie Datenbanktransaktionen aufweist:

- Änderungen sind atomar, werden also nur vollständig oder gar nicht ausgeführt.
- Threads sehen zu jedem Zeitpunkt einen konsistenten Zustand.
- Und schließlich sind nebenläufige Threads voneinander isoliert, d. h. Zustandsänderungen eines Threads sind in einem anderen Thread nicht sichtbar, bis die Transaktion abgeschlossen ist.

Damit sind die Eigenschaften Atomarität, Konsistenz und Isoliertheit von ACID-Transaktionen umgesetzt, nicht jedoch die Dauerhaftigkeit. Der transaktionale Zugriff auf Werte wird in Clojure mit einem speziellen Referenz-Typ realisiert. Mit **ref** wird eine neue Referenz auf einen Wert erzeugt. Der referenzierte Wert ist wie bei Agenten mit der Funktion **(deref <ref>)** bzw. dem Makro **@<ref>** zugänglich. Wird die Referenz verändert, so wird dabei nicht der Wert verändert, sondern die Referenz zeigt nach der Änderung auf einen anderen Wert.

Veränderungen von Referenzen sind mit den Funktionen **ref-set** und **alter** nur innerhalb einer Transaktion zulässig, die durch einen **dosync**-Block gestartet wird. Die Änderungen an einer Referenz werden erst dann in anderen Threads sichtbar, wenn der **dosync**-Block beendet wird.

Wurden Referenzen seit dem Start des **dosync**-Blocks durch einen anderen Thread verändert, so wird die Transaktion abgebrochen und neu gestartet. Die Zahl der Wiederholungen wird durch eine implementierungsabhängige Konstante beschränkt. Da Code in einer Transaktion wiederholt werden kann, sollte Code mit Nebeneffekten nicht innerhalb eines **dosync**-Blocks verwendet werden, da auch dieser Code wiederholt wird.

Aus diesem Grund werden auch **send**-Operationen an Agenten in einem **dosync** erst nach erfolgreichem Abschluss der Transaktion ausgeführt (**release-pending-sends** hat innerhalb eines **dosync** keinen Effekt) und das Warten auf einen Agenten mit **await** innerhalb eines **dosync**-Blocks wird mit einer Exception abgebrochen.

Als Beispiel für STM soll die Überweisung zwischen zwei Konten dienen. Zunächst werden zwei **ref** angelegt und mit 100 als Wert initialisiert, die zwei Konten abbilden:

```
>> (def konto-a (ref 100))
>> (def konto-b (ref 100))
```

Die Überweisung eines Betrags zwischen zwei Konten kann dann durch die Funktion **überweise** mit Hilfe von **ref-set** implementiert werden:

```
>> (defn überweise [quelle ziel betrag]
     (dosync
      (ref-set quelle (- @quelle betrag))
      (ref-set ziel   (+ @ziel betrag))))
```

Die Funktion **ref-set** weist der Referenz einen neuen Wert zu. Alternativ kann die Funktion **alter** verwendet werden. Sie übernimmt die zu verändernde Referenz als erstes Argument und als zweites eine Funktion, die aus dem aktuellen Wert der Referenz und optionalen Argumenten den neuen Wert der Referenz berechnet. Damit hat **alter** einen ähnlichen Aufbau wie **send** (s. Agenten):

```
>> (defn überweise [quelle ziel betrag]
     (dosync
      (alter quelle #(- % betrag))
      (alter ziel   #(+ % betrag))))
```

Mit

```
>> (überweise konto-a konto-b 10)
```

können 10 Einheiten zwischen den Konten transferiert werden:

```
>> @konto-a ==> 90
>> @konto-b ==> 110
```

Wichtig ist, dass die Änderungen an **konto-a** und **konto-b** entweder gemeinsam in anderen Threads sichtbar werden oder gar nicht.

Das letzte Beispiel soll das Zusammenspiel von Agenten und STM zeigen. Wir simulieren ein Abendessen, bei dem nur ein Messer und eine Gabel vorhanden sind, die beide für das Essen benötigt werden.

Jeder der anwesenden Gäste verhält sich gleich: Sind Messer und Gabel unbenutzt, versucht der Gast, beide in seinen Besitz zu bringen, einen Gang zu essen und dann Messer und Gabel abzulegen. Das wiederholt jeder Gast so oft, bis er die drei Gänge des Abendessens eingenommen hat. Messer und Gabel können von je höchstens einem Gast benutzt werden.

Der Zustand eines Gastes wird durch seinen Namen, einen „Ref“-Verweis auf die Zahl der eingenommenen Gänge sowie je einen „Ref“-Verweis auf den aktuellen Besitzer von Messer und Gabel beschrieben. Alle Gäste nutzen dieselben Referenzen auf Messer bzw. Gabel. Wir stellen den Gast-Zustand mit Clojures StructMaps dar:

```
>> (defstruct gast :name :gänge :messer :gabel)
```

Damit können wir Funktion schreiben, mit denen das oben beschriebene Verhalten eines Gastes implementiert wird: Das Besteck ist dann frei, wenn kein Agent als Besitzer eingetragen ist:

```
>> (defn besteck-frei? [state]
(not (or @(state :gabel) @(state :messer))))
```

Der Vorgang des Besteckaufnehmens trägt den Agenten als Besitzer von Messer und Gabel innerhalb einer Transaktion ein:

```
>>(defn besteck-nehmen [state]
    (println (state :name) "Messer, Gabel?")
    (dosync
     (ref-set (state :gabel) *agent*)
     (ref-set (state :messer) *agent*)))
```

Das Aufnehmen des Bestecks war erfolgreich, wenn der Agent als Besitzer von Messer und Gabel eingetragen ist:

```
>> (defn besteck-meins? [state]
     (and (= @(state :gabel) *agent*)  ;; @ … : Wert der Ref
          (= @(state :messer) *agent*)))
```

Der Vorgang des Besteckablegens setzt den Besitzer von Messer und Gabel in einer Transaktion auf `nil`:

```
>> (defn besteck-ablegen [state]
     (dosync
      (ref-set (state :gabel) nil)
      (ref-set (state :messer) nil)))
```

Ein Gast ist mit dem Essen fertig, wenn er alle drei Gänge eingenommen hat:

```
>> (defn essen-fertig? [state]
  (>= @(state :gänge) 3))
```

Der Essvorgang beschäftigt den Agenten für eine Zeit von 200 ms und erhöht dann die Zahl der eingenommenen Gänge:

```
>> (defn essen [state]
     (println (state :name) "Messer, Gabel, Essen !!!")
     (Thread/sleep 200)
     (println (state :name) "Hmm")
     (dosync
       (alter (state :gänge) (fn [_] (+ 1 _)))))
```

Damit sind alle Bausteine vorhanden, mit denen eine Agenten-Funktion für das Verhalten der Gäste geschrieben werden kann:

```
>> (defn verhalten [state]
     (loop []
       (dosync                ;; "besteck-frei?" und besteck-nehmen
        (if (besteck-frei? state) ;; müssen in einer gemeinsamen
          (besteck-nehmen state)));; Transaktion ablaufen
       (if (besteck-meins? state)
         (do
           (essen state)
           (besteck-ablegen state)))
       (if (essen-fertig? state)
          state
         (recur))))
```

Um alles zusammenzubauen, benötigen wir noch das gemeinsam genutzte Besteck

```
>> (def das-messer (ref nil))
>> (def die-gabel (ref nil))
```

und die Gäste, die alle das gleiche Besteck nutzen

```
>> (defn neuer-gast [name]
     (agent (struct gast name (ref 0) das-messer die-gabel)))
>> (def gäste  (map neuer-gast
                 ["Kant" "Platon" "Wittgenstein" "Sokrates"]))
```

Das Essen beginnt mit

```
>> (dorun (map (fn [_] (send _ verhalten)) gäste))
```

und mit

```
>> (map (fn [gast] (list (@gast :name) @(@gast :gänge))) gäste)
```

bekommen wir jederzeit Einblick in den Fortgang des Abendessens.

Anmerkungen zu unserer Implementierung:

- Die Implementierung von **`verhalten`** ist nicht sonderlich effizient, jeder Agent fragt den Zustand der Besteck-Ressourcen ständig ab. Ein Blick auf die CPU-Auslastung bei der Ausführung bestätigt das.
- Die Gäste nehmen Messer und Gabel gleichzeitig auf. Damit kann das Besteck wie eine einzige Ressource behandelt werden und damit kann auch keine Verklemmung (Deadlock) auftreten. Möglich wird das durch die **`dosync`**-Transaktion.
- Da jeder Gast nach einer festen Anzahl von Gängen mit dem Essen aufhört, verhungern selbst langsame Gäste nicht. Diese Implementierung garantiert aber nicht, dass das Besteck den Gästen „fair“ zugeteilt wird.

Zusammenfassung

Beschäftigt man sich zum allerersten Mal mit einem Lisp-Derivat, wirkt Clojure ungewohnt – vor allem die Syntax unterscheidet sich deutlich von der üblicher Mainstream-Sprachen wie Java, C#, C/C++ oder auch Scala. Man gewöhnt sich daran jedoch nicht nur schnell, sondern lernt sie insbesondere durch die Unterstützung für Makros sogar sehr schätzen.

Mit seiner perfekten Java-Integration und dem sich dadurch ergebenden Ökosystem von Bibliotheken, den verfügbaren Entwicklungsumgebungen und der sehr aktiven Community, aber auch mit der in diesem letzten Teil unserer Artikelserie vorgestellten Concurrency-Unterstützung ist Clojure eine überaus praktische Sprache, die auch für den Einsatz in gemischten Projekten – Stichwort „Polyglottes Programmieren“ – hervorragend geeignet ist.

Links

[clojure1.1.0] Clojure-Quellen und jar-Archiv, **`http://clojure.googlecode.com/files/clojure-1.1.0.zip`**

[ClojureCore] API for clojure.core, **`http://richhickey.github.com/clojure/clojure.core-api.html#clojure.core/letfn`**

[Hick09] R. Hickey, Persistent Data Structures and Managed References, Folien zum Vortrag auf der QCon, 2009, **`http://qconlondon.com/london-2009/file?path=/qcon-london-2009/slides/RichHickey_PersistentDataStructuresAndManagedReferences.pdf`**

[NeTi10a] B. Neppert, St. Tilkov, Clojure: Funktional, parallel, genial – Teil 1: Überblick, in: JavaSPEKTRUM, 2/2010

[NeTi10b] B. Neppert, St. Tilkov, Einführung in Clojure – Teil 2, in: JavaSPEKTRUM, 3/2010

Burkhard Neppert ist Senior Consultant bei der innoQ Deutschland GmbH. Er beschäftigt sich dort mit Konzeption und Entwicklung unternehmenskritischer Systeme, vorwiegend in Mainstream-Sprachen wie Java oder C++.
E-Mail: burkhard.neppert@innoq.com

Stefan Tilkov ist Geschäftsführer und Principal Consultant bei der innoQ Deutschland GmbH, wo er sich mit Architekturen für verteilte Systemlandschaften beschäftigt. Sein aktuelles Hauptinteresse liegt auf dynamischen Sprachen und dem Einsatz von RESTful HTTP für Integrationsarchitekturen.
E-Mail: stefan.tilkov@innoq.com