Seminar Programmiersprachen und Programmiersysteme

Software Transactional Memory in Scala

Nicolas Günther

29. Januar 2010

Betreuer: Prof. Dr. Frank Huch

Inhaltsverzeichnis

1 Einleitung

Mit der fortschreitenden Verbreitung von Multiprozessor-Systemen, ergeben sich heute für Programmierer neue Herausforderungen. Um die Fähigkeiten neuer Rechner auszunutzen, reicht es nicht mehr aus sequentiellen performanten Code zu schreiben, es wird immer wichtiger Programme zu parallelisieren.

Klassische Konzepte zur nebenläufigen Programmierung vermeiden Probleme der nebenläufigen Ausführung meist durch gegenseitigen Ausschluss von Prozessen in kritischen Codebereichen. Es ist oft schwer mit diesen Konzepten korrekte nebenläufige Programme zu schreiben, die auch noch lesbar und leicht zu warten sind.

Das Konzept des *Software Transactional Memory* verspricht die Schwierigkeiten beim Entwurf nebenläufiger Programme zu reduzieren. Typische Probleme der nebenläufigen Programmierung sollen konzeptionell ausgeschlossen werden:

- *Deadlock*: Im STM-Konzept gibt es keine Primitive zum Sperren des Zugriffs auf Variablen, stattdessen basiert die nebenläufige Programmierung auf *Transaktionen*.
- *Priority Inversion*: Ohne Mechanismen zum Sperren von Objekten kann es nicht passieren, das ein Prozess eine Sperre auf ein Objekt hält und dadurch einen höher priorisierten Prozess an der Ausführung hindert.

Es existieren STM-Implementierung in einer Vielzahl von Programmiersprachen (siehe [5]). Thema dieser Ausarbeitung ist die STM-Implementierung von Daniel Spiewak in der Programmiersprache Scala (siehe [4]).

In Kapitel 2 werden grundlegende Konzepte von STM behandelt, in Kapitel 3 die zum Verständnis der Bibliothek nötigen Konzepte von Scala erläutert. Anschliessend werden in Kapitel 4 die Implementierungsdetails von STM in Scala beleuchtet.

2 Software Transactional Memory

In STM werden nebenläufig auszuführende Codesequenzen als atomar gekennzeichnet. Um zu bearbeitende Speicherbereiche vor Inkonsistenzen zu schützen, die durch nebenläufige Ausführung mehrerer Prozesse entstehen können, wird nicht das klassische Konzept des pessimistischen Sperrens angewandt. Statt dessen spricht man in STM von der *optimistischen* Ausführung von Transaktionen.

2.1 Transaktionen

Eine *Transaktion* ist eine endliche Sequenz von Anweisungen, die von einem einzelnen Thread ausgeführt wird und folgende Eigenschaften erfüllt:

1. *Serialisierbarkeit:* Transaktion scheinen seriell ausgeführt zu werden. Das heisst, die Anweisungen einer Transaktion scheinen niemals durch die Anweisungen einer anderen Transaktion unterbrochen zu werden.

2. *Atomarität:* Während ihrer Ausführung, macht eine Transaktion eine Reihe von Speichermanipulationen. Am Ende der Transaktion wird diese entweder *commited*, das heisst die Speichermanipulationen werden für andere Prozesse sichtbar, oder *aborted*, dass heisst die Speicheränderungen werden verworfen und die Transaktion erneut ausgeführt. In einer Transaktionen dürfen nur seiteneffektfreie Funktionen aufgerufen werden, da sonst eine Wiederherstellung des ursprünglichen Zustands nicht möglich ist.

Shavit und Touitou [2] haben zur Definition von Transaktionen in Programmiersprachen folgende Syntax vorgeschlagen:

```
BeginTransaction
  ...
EndTransaction
```

In aktuellen Implementierungen werden Transaktionen meist als Blöcke in folgender oder ähnlicher Form definiert, manchmal mit einer zusätzlichen Bedingung `cond`:

```
atomic (cond) {
  ...
}
```

In vielen Implementierungen ist die Komposition von Transaktionen möglich, beispielsweise in [1]:

```
atomic {
  ...
} orElse {
    ...
}
```

Hier führt der Thread zuerst den ersten Block aus. Falls die Transaktion abgebrochen wird, führt der Thread nach dem Verwerfen der Speicheränderungen

den zweiten Block aus. Kommt es auch hier zu einem Abbruch, wird das gesamte Statement erneut ausgeführt. In der Regel wird eine Transaktion erst erneut ausgeführt, wenn sich im entsprechenden Kontext etwas geändert hat.

Meist ist es auch möglich eine Transaktion mittels der Anweisung `retry` manuell abzubrechen:

```
atomic {
  if (b) {
    retry;
  }
  ...
}
```

Idealerweise wird die Transaktion dann erst wieder ausgeführt, wenn sich der Wert von `b` geändert hat.

2.2 Implementierungen

Im folgenden werden die wesentlichen generellen Unterschiede aktueller Implementierungen dargestellt.

Man unterscheidet zwischen statischen und nicht statischen Transaktionen. Bei statischen Transaktionen wird der zu schützende gemeinsame Speicherbereich explizit angegeben. Das heisst einer Transaktion werden z.B. die zu schützenden Variablen als Parameter übergeben. Man spricht von nicht statischen Transaktionen, falls der entsprechende Kontext nicht angegeben werden muss bzw. nicht bekannt ist. Eine solche STM-Implementierung muss die zu schützenden Speicherbereiche also dynamisch bestimmen und überwachen.

Es gibt Implementierungen, die die Komposition von Transaktionen ermöglichen (siehe 2.1). Andere Implementierungen erlauben nur einfache Transaktionen.

Einige Implementierungen erlauben das Verschachteln von Transaktionen in folgender oder ähnlicher Form:

```
atomic {
  atomic{
    ...
  }
  atomic {
    ...
  }
  ...
}
```

Solche Implementierungen können sich unterschiedlich verhalten, wenn es zum Abbruch einer inneren Transaktion kommt. Beispielsweise kann dies den Abbruch der umgebenden Transaktion bedeuten, oder aber nur die Wiederholung der abgebrochenen inneren Transaktion.

In einer *lock-freien* STM-Implementierung werden Transaktionen grundsätzlich gestartet und zu Ende ausgeführt. Dies geschieht unter der Annahme, das keine Konflikte auftreten werden. Erst am Ende einer Transaktion wird dann geprüft, ob sich der Speicher in einem konsistenten Zustand befindet, oder ob die Transaktion wiederholt werden muss. In einer *lock-basierten* Implementierung werden zu schützende Speicherbereiche gelockt, so das sie zu jedem Zeitpunkt nur von genau einer Transaktion manipuliert werden können.

3 Scala

Scala ist sowohl eine objektorientierte, als auch eine funktionale Sprache. Die Typisierung ist statisch, aber es ist oft möglich auf Typannotationen zu verzichten, wobei die Typen dann mittels Typinferenz hergeleitet werden. Ziel bei der Entwicklung von Scala war es, aufbauend auf einem Kern von wenigen leistungsfähigen Konstrukten, das Erstellen von Bibliotheken zu ermöglichen, die die Sprache bezüglich der jeweiligen Bedürfnisse anpassen bzw. erweitern. Diese Bibliotheken sollen sich nahtlos in die Sprache integrieren. Erreicht werden soll dies durch die Kombination von objektorientierten und funktionalen Konzepten.

3.1 Klassen

Die Syntax zur Definition von Klassen in Scala entpricht im wesentlichen der Syntax von Java:

```
class Rational(val n: Int, val d: Int) {
  val view: String = n + "/" + d
  override def toString = view
  def this(n: Int) = this(n, 1)
  def +(that: Rational): Rational =
    new Rational(
      n * that.d + that.n * d,
      d * that.d
    )
}
```

Die Klasse Rational repräsentiert eine rationale Zahl, definiert durch ihren Nenner `n` und ihren Zähler `d`. Die Konstruktorargumente werden direkt als Parameter der Klasse angegeben. Typen werden mit einem Doppelpunkt nach einem Bezeichner angegeben. Zusätzlich ist ein Konstruktor zum bequemeren Erzeugen einer rationalen Zahl aus einer ganzen Zahl angegeben.

Sonderzeichen wie z.B. `+` sind in Scala als Methodennamen erlaubt. Man hat so die Möglichkeit Operatoren zu definieren, welche in Infix-Notation verwendet werden können:

```
var a = new Rational(2)
var b = new Rational(2,3)
var c = a + b //enspricht a.+(b)
```

Die Bindungsstärke und Assoziativität von Operatoren hängt von dem Sonderzeichen ab, mit dem der Name der Methode endet.

3.2 Funktionen

Funktionen sind in Scala Werte erster Klasse. Das Definieren von Funktionen als Merkmale einer Klasse, d.h als Methoden, ist in Scala nicht die einzige Möglichkeit Funktionen zu erzeugen. Funktionen können auch lokal innerhalb anderer Funktionen definiert werden. Darüberinaus können Funktionen anonym als Funktionsliterale ohne Namen definiert werden:

```
(x: Int) => x % 2 == 0
```

Funktionen lassen sich beliebig an Variablen binden und können als Argumente an andere Funktionen übergeben werden.

Wie aus funktionalen Sprachen bekannt, erlaubt Scala das Curryfizieren von Funktionen:

```
def sum(x: Int)(y: Int) = x + y;
```

Statt die Parameter in Form einer Liste anzugeben, sind die Parameter hier in der curryfizierten Variante mit mehreren Parameterlisten angegeben. Mittels partieller Applikation kann man nun aus `sum` eine Funktion gewinnen, die eine ganze Zahl inkrementiert:

```
def inc = sum(1)_
```

Der Unterstrich ist hier ein Platzhalter für das fehlende zweite Argument.

3.3 Implizite Konvertierungen und Parameter

Im Falle von potentiellen Typfehlern prüft der Scala-Compiler, ob sich im aktuellen Scope eine Funktion befindet, die im falschen Typkontext eingesetzte Objekte auf den erwarteten Typ abbildet. Solche Funktionen müssen mit dem Schlüsselwort `implicit` gekennzeichnet sein.

Die Klasse `Rational` aus 3.1 besitzt einen Konstruktor zur einfacherern Erzeugung einer rationalen Zahl aus einer ganzen Zahl:

```
val x: Rational = new Rational(3)
```

Folgende *implizite Konvertierung* ermöglicht eine natürlichere Syntax zur Erzeugung rationaler Zahlen aus ganzen Zahlen:

```
implicit def int2rational(x: Int): Rational = new Rational(x)
```

Befindet sich diese Funktion im aktuellen Scope, behebt der Compiler potentielle Typfehler in Situationen wie:

```
val x: Rational = 3
3 + new Rational(1,2)
```

Man kann sich diese implizite Konvertierung als eine Erweiterung der Klasse `Int` um eine Methode `+` zur Addition mit rationalen Zahlen vorstellen.

In der letzten curryfizierten Argumentliste einer Funktion kann man Argumente mit dem Schlüsselwort `implicit` kennzeichnen: [1]

```
def qsort[T](list: List[T])(implicit ord: T => Ordered[T]) =
  list match {
    case Nil => Nil
    case x :: xs => qsort(list.filter(_ < x))
                    ::: List(x)
                    ::: qsort(list.filter(_ > x))
  }
```

Die Funktion `qsort` sortiert eine Liste mit dem Quicksort-Algorithmus. Die Elementtypen der Liste sind hierbei generisch mit Typparameter `T`.

Das Kennzeichnen der letzten Parameterliste mit `implicit` verlangt die Existenz einer impliziten Konvertierung `ord`, die Elemente des Typs `T` auf Elemente des Typs `Ordered[T]` abbildet. `Ordered` ist eine in Scala vordefinierte abstrakte Klasse, die typische Vergleichsoperationen vorschreibt. Ähnlich wie Typklassen in Haskell erlauben implizite Parameter zusätzliche

[1] Die Notation `(_ < x)` ist eine abkürzende Schreibweise für die Closure `(a: Int) => a < x`

Anforderungen an Typparameter zu stellen (siehe [3]). Befindet sich nun eine geeignete implizite Konvertierung im Scope, können wir die Funktion `qsort` ohne Angabe des zweiten Arguments verwenden:

```
qsort(List(7,6,5,4,3,2,1))
```

oder selbst eine Funktion zur Konvertierung angeben:

```
qsort(List("h","e","l","l","o"))(myord)
```

4 Software Transactional Memory in Scala

Die in dieser Ausarbeitung betrachtete Bibliothek zur Erweiterung von Scala um das Konzept des STM wurde entwickelt von Daniel Spiewak (siehe [4]). Im folgenden wird diese Bibliothek mit *scala-stm* bezeichnet.

4.1 Referenzen

Das Konzept von scala-stm basiert auf Referenzen. Jede Variable, die vor einem inkosistenten Zustand durch nebenläufige Manipulation geschützt werden soll, muss als eine Referenz gekapselt werden. Aus einer Referenz kann immer gelesen werden, in eine Referenz geschrieben werden kann nur innerhalb einer Transaktion.

Wird innerhalb einer Transaktion in eine Referenz geschrieben, ist der neue Wert der Referenz vorerst nicht global sichtbar, solange bis die Transaktion erfolgreich commited.

Eine Referenz ist in scala-stm ein Objekt vom Typ `Ref`:

```
class Ref[T](v: T) {
  private var contents_ = (v, 0)
  def get(implicit c: Context) = c retrieve this
  def :=(v: T)(implicit c: Transaction) {
    c.store(this)(v)
  }
  ...
}
```

Ein solches Objekt enthält im wesentlichen das Tupel `contents_`, bestehend aus dem Wert `v` und der initialen Revisionsnummer 0. Die Methoden `get` und `:=` zum Lesen bzw. Schreiben des Wertes einer Referenz, fordern durch implizite Parameter einen `Context` bzw. eine `Transaction` im aktuellen Scope.

Um Referenzen wie normale Variablen lesen zu können, ohne bei jedem Zugriff die Methode `get` aufrufen zu müssen gibt es ein globale implizite Konvertierung:

```
implicit def refToValue[T](ref: Ref[T])(implicit c: Context)
  = ref.get(c)
```

4.2 Kontext

Je nach Situation in der eine Referenz manipuliert wird, muss die Referenz in einem unterschiedlichen *Kontext* angesprochen werden. So unterscheidet sich beispielsweise der Kontext beim Ändern des Wertes einer Referenz innerhalb zweier unterschiedlicher Transaktionen. In scala-stm wird zwischen einem globalen Live-Kontext und dem lokalen Transaktions-Kontext unterschieden: `Context` ist hier eine abstrakte Klasse und `LiveContext` ein Singleton-

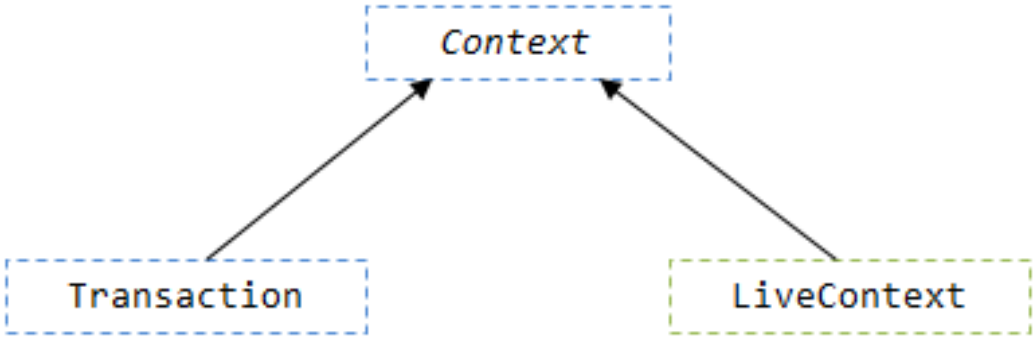

Abbildung 1: Kontext-Hiererchie aus [5]

Objekt: [2]

```
abstract class Context private[stm] () {
  private[stm] def retrieve[T](ref: Ref[T]): T
}
implicit object LiveContext extends Context {
  private[stm] def retrieve[T](ref: Ref[T]) = ref.value
}
```

Die Annotation `private[stm]` an der Klasse `Context` bewirkt, das ein Erweitern der Klasse ausserhalb des Pakets `stm` nicht möglich ist.

[2]Eine mit dem Schlüsselwort `object` definierte Klasse entspricht nicht einer statischen Klasse, sondern dem Singleton-Pattern.

Jede Transaktion hat einen eigenen `Transaction`-Kontext, welcher bei Manipulationen von Referenzen verwendet wird. `LiveContext` wird nur ausserhalb von Transaktionen verwendet, wenn kein anderer Kontext verfügbar ist. Um das Schreiben von Referenzen ausserhalb von Transaktionen zu unterbinden, ist der Argumenttyp der Methode `:=` in der Klasse `Ref` auf den Typ `Transaction` beschränkt (siehe 4.1).

Die Klasse `Transaction` unterscheidet sich vom `LiveContext` vor allem durch eine Map, die vom Live-Kontext abweichende Referenzen hält. Ausserdem enthält `Transaction` eine Map, welche die Revisionsnummern der innerhalb der Transaktion verwendeten Referenzen hält.

```
final class Transaction private[stm] (val rev: Int)
extends Context {
  private[stm] val world = mutable.Map[Ref[Any], Any]()
  private val version = mutable.Map[Ref[Any], Int]()
  private val writes = mutable.Set[Ref[Any]]()

  private[stm] def retrieve[T](ref: Ref[T]) {
    val castRef = ref.asInstanceOf[Ref[Any]]
    if(!world.contains(castRef)) {
      val (value, refRev) = ref.contents
      world(castRef) = value
      if (!version.contains(castRef)) {
        version(castRef) = refRev
      }
    }
    world(castRef).asInstanceOf[T]
  }

  private[stm] def store[T](ref: Ref[T])(value: T) {
    val castRef = ref.asInstanceOf[Ref[Any]]
    if(!version.contains(castRef)) {
      version(castRef) = ref.rev
    }
    world(castRef) = value
    writes += castRef
  }
  ...
}
```

Um nicht alle Referenzen des Live-Kontexts kopieren zu müssen, geschieht dies erst bei Bedarf. Beim ersten Lesen einer Referenz mit `retrieve` im

Kontext einer Transaktion, wird die Referenz in die Map `world` eingetragen und die globale Revisionnummer der Referenz zum Zeitpunkt des Lesezugriffs wird in die Map `version` eingetragen.

Beim Schreibzugriff innerhalb der Transaktion wird der neue Wert mit der Referenz als Schlüssel in der Map `world` gespeichert und als Versionsnummer wird wieder die ursprüngliche Revisionsnummer in die Map `version` eingetragen.

4.3 Validierung und Commit

Am Ende einer Transaktion wird die Methode `commit` der Klasse `Transaction` aufgerufen:

```
private[stm] def commit() {
  if(world.size > 0) {
    CommitLock.synchronized {
      val back = world.foldLeft(true) { (success, tuple) =>
        val (ref, _) = tuple
        success && ref.rev == version(ref)
      }

      if(back) {
        for(ref <- writes) {
          ref.contents = (world(ref), rev)
        }
      }
      back
    }
  } else true
}
```

Der erste Schritt des Commitments ist die Validierung. Entsprechen die Revisionsnummern aller innerhalb der Transaktion verwendeten Referenzen den Revisionsnummern welche vor den jeweiligen Zugriffen in der Map `world` eingetragen wurden, können die transaktionslokalen Änderungen an Referenzen in den globalen Live-Kontext übernommen werden. Die Revisionsnummern der Referenzen im Live-Kontext werden dabei auf die eindeutige Revisionsnummer der Transaktion `rev` gesetzt. Wenn sich die lokalen Revisionsnummern von denen in `world` gespeicherten unterscheiden, werden die transaktionslokalen Änderungen verworfen und `commit` liefert `false` zurück. Dies führt dazu, das die Transaktion erneut ausgeführt wird.

4.4 Das Schlüsselwort `atomic`

Das Schlüsselwort `atomic` dient der Erzeugung und der Durchführung von Transaktionen auf Benutzerebene (siehe 2.1). In scala-stm ist `atomic` als Funktion implementiert, welche sich bei Verwendung der Bibliothek global im Scope befindet:

```
def atomic[A](f: (Transaction)=>A): A = atomic(true)(f)

def atomic[A](cond: =>Boolean)(f: (Transaction)=>A) {
  def attemptTransact(): A {
    if(cond) {
        val trans = new Transaction(rev)
        try {
          val result = f(trans)
          if(trans.commit()) result else attemptTransact()
        } catch {
          case RetryMessage => {
            val block = new AnyRef
            for((ref, _) <- trans.world) {
              ref.registerBlock(block)
            }
            block.wait()
            for((ref, _) <- trans.world) {
              ref.deregisterBlock(block)
            }
            attemptTransact()
         }
         case FailureException(e) => {
           throw e
         }
         case _ => attemptTransact()
       }
    } else null.asInstanceOf[A]
  }
  attemptTransact()
}
```

Die innere Funktion `attemptTransact` erzeugt eine Transaktion mit Revisionsnummer `rev`, eine global verfügbare Funktion, die eindeutige Revisionsnummern vergibt. Das Konstrukt `atomic` übernimmt als Parameter eine Funktion `f`, diese entspricht dem als Transaktion auszuführenden Code. Die-

se Funktion wird dann mit der neu erzeugten Transaktion `trans` als Kontext ausgeführt. Anschliessend wird `trans.connmit` aufgerufen. Im Falle fehlgeschlagener Validierung wird `attemptTransact` neu aufgerufen, sonst ist die Transaktion damit beendet.

Ausnahmen vom regulären Ablauf einer Transaktion werden mittels Exceptions abgehandelt. Exceptions vom Typ `FailureException` werden abgefangen und nach aussen propagiert. Bis auf Exceptions vom Typ `RetryMessage` führen alle anderen Exceptions zu einer Wiederholung der Transaktion.

Die `RetryMessage` ermöglicht dem Programmierer mittels der global verfügbaren Funktion `retry`, manuell Transaktionen neu zu starten:

```
private case object RetryMessage extends RuntimeException
def retry()(implicit t: Transaction) {
  throw RetryMessage
}
```

Im Falle einer durch `retry` abgebrochenen Transaktion, wird die Transaktion bei allen in der Transaktion verwendeten Referenzen mittels `registerBlock` registriert und wartet dann mit einem Aufruf von `wait`. Sobald eine dieser Referenzen sich ändert, wird die Transaktion mit `notify` aufgeweckt. Daraufhin macht die Transaktion alle Registrierungen auf Referenzen wieder rückgangig und wird mit `attemptTransact` neu gestartet.

4.5 Beispiel

Anhand eines einfachen Beispiels soll die Verwendung der Bibliothek demonstriert werden:

```
object EasyTest {
  import Transaction._

  val count = new Ref[Int]

  def main(args: Array[String]) {
    val left = new Thread {
      override def run() {
        atomic {
          countImpl(_)
        }
      }
    }
```

```
    val right = new Thread {
      override def run() {
        atomic {
          countImpl(_)
        }
      }
    }

    left.start()
    right.start()

    left.join()
    right.join()

    val result: Int = count
    printf("Result: %d%n", result)
  }

  private def countImpl(implicit t: Transaction) {
    for (i <- 0 until 10) {
      val i: Int = count
      Thread.sleep((Math.random * 50).intValue)

      count := i + 1
    }
  }
}
```

Das Beispiel besteht aus zwei Transaktionen, die wiederholt die Referenz `count` beschreiben. Die Funktion `countImpl` erwartet durch einen impliziten Parameter einen Transaktions-Kontext, daher kann sie in einem Thread in dem `atomic`-Block aufgerufen werden. Durch zufälliges Pausieren der Threads werden Konfliktsituationen provoziert. Die Zuweisung `count := i + 1` erfolgt mittels der in 4.1 beschriebenen Methode `:=` der Klasse `Ref`, welche nur innerhalb einer Transaktion ausgeführt werden kann.

5 Fazit

Daniel Spiewak ist es gelungen die Konzepte von STM auf natürliche Weise in Scala zu integrieren. Durch die Verwendung von Funktionen höherer Ordnung und impliziten Konvertierungen, schafft er es Referenzen, Transaktions-

Kontexte und die atomic-Anweisung, wie feste Bestandteile der Sprache wirken zu lassen.

Das Kernproblem bei der Implementierung von STM in Sprachen, die nicht rein funktional sind, ist es mit eventuellen Seiteneffekten umzugehen. In scala-stm ist es Aufgabe des Programmierers, dafür zu sorgen, dass innerhalb von Transaktionen keine Funktionen mit Seiteneffekten verwendet werden. Dies ist in Scala zwar möglich, aber nicht immer leicht zu durchschauen. Aktuell diskutiert wird ein Compiler-Plugin, das zur Compilezeit testet, ob rein funktionaler Code vorliegt. Dies würde die Verwendung von scala-stm deutlich sicherer machen.

In scala-stm ist die Komposition und das Verschachteln von Transaktionen nicht möglich. Konstrukte wie eine orElse-Anweisung, wie aus Implementierungen für andere Sprachen bekannt, wurden nicht berücksichtigt. Da beim Commit von Transaktionen die Revisionsnummern von Referenzen immer mit denen im globalen Live-Kontext verglichen werden, sind verschachtelte Transaktionen nicht möglich. Hier könnte eine Baumstruktur zur Überwachung von Revisionsnummern Abhilfe schaffen.

Eine Weitere Unzulänglichkeit in scala-stm ist der globale Lock, der die gemeinsamen Daten beim Commit von Transaktionen schützt. Auch Transaktionen die auf gänzlich unterschiedlichen Referenzen arbeiten, können bei zeitnahem Commit zum Warten gezwungen werden. Dies könnte sich als Flaschenhals bezüglich der Performance erweisen.

Literatur

[1] N. Shavit M. Herlihy. *The Art Of Multiprocessor Programming.* Morgan Kaufman, 2008.

[2] D. Touitou N. Shavit. Software transactional memory. Technical report, MIT, Tel-Aviv University, 1995.

[3] M. Odersky. Poor man's type classes. Talk at IFIP WG 2.8, 2006.

[4] Daniel Spiewak. Software transactional memory in scala, 2008. http://www.codecommit.com/blog/scala/software-transactional-memory-in-scala.

[5] Wikipedia. Software transactional memory, January 2010. http://en.wikipedia.org/software_transactional_memory.