Konkurrente Programmierung in Haskell
Software Transactional Memory

Bastiaan Zapf

6. Dezember 2005

Zusammenfassung

Probleme des traditionellen Ansatzes (Deadlocks, Prioritätsinversion) werden aufgezeigt. Eine kurze Einführung in STM wird gegeben. Skiplists werden als STM-verträgliche Alternative zu Rot/Schwarz-Bäumen eingeführt. Die Beziehung von STM-Semantik zu wichtigen Sprachfamilien (Imperativ, Funktional) wird aufgezeigt, sowie der begründete Verdacht geäußert, dass STM in Hardware implementierbar ist. Kontraste von Haskell-Typklassen zu OOP-Typklassen werden beschrieben. Eine kurze Einführung in das Konzept der Monaden wird gegeben. Die durch [HHMPJ05] begründete monadische Umschreibung von STM-Prozessen wird erläutert, und anhand von zwei Beispielen (Multicast-Kanäle, dinierende Philosophen) erklärt.

Inhaltsverzeichnis

1 Einführung

1.1 Sojourner - Rekonstruktion einer Panne

Der Mars-Rover Sojourner arbeitete erfolgreich, bis auf gelegentliche Abstürze, die bis zum nächsten Funkkontakt nicht behoben werden konnten.

> The failure was identified by the spacecraft as a failure of the bc_dist task to complete its execution before the bc_sched task started. The reaction to this by the spacecraft was to reset the computer. This reset reinitializes all of the hardware and software. It also terminates the execution of the current ground commanded activities. No science or engineering data is lost that has already been collected (the data in RAM is recovered so long as power is not lost). However, the remainder of the activities for that day were not accomplished until the next day.
> [. . .]
> The failure turned out to be a case of priority inversion

([Jon97])

Abb. 1 stellt das Problem grafisch dar. Der Thread mit derf niederigsten Priorität wird nicht abgebrochen, wenn der mit der höchsten auf den Bus zugreifen will. Wird der mit der niedrigsten jedoch von einem lang dauernden Thread mittlerer Priorität unterbrochen, setzt ein Watchdog den Roboter in einen Fehlermodus.

Strenggenommen ein Hardwareproblem, zeigt sich jedoch, dass viele Hardwareblockaden günstig umgangen werden können, indem Protokolle feinkörnig organisiert werden und die Priorisierung auf höherer Ebene realisiert wird. Beispiel: *Asynchronous Transfer Mode*

In Multithreadingsystemen könnte STM eine entscheidende Vereinfachung der Implementierung anbieten.

1.2 Deadlock

Philosophen sitzen an einem Tisch. Zwischen jeweils zwei Philosophen liegt eine Gabel. Jeder Philosoph denkt eine Weile, wird dann hungrig, und möchte unter Zuhilfenahme zweier Gabeln essen. Dazu nimmt er erst eine, dann eine zweite Gabel. Ohne weitere Vorkehrungen kann folgende Situation eintreten: Jeder Philosoph hält eine Gabel in der linken Hand, und wartet darauf, dass der Philosoph rechts neben ihm seine Gabel wieder hinlegt. So können alle Philosophen am gedeckten Tisch zu Tode hungern.

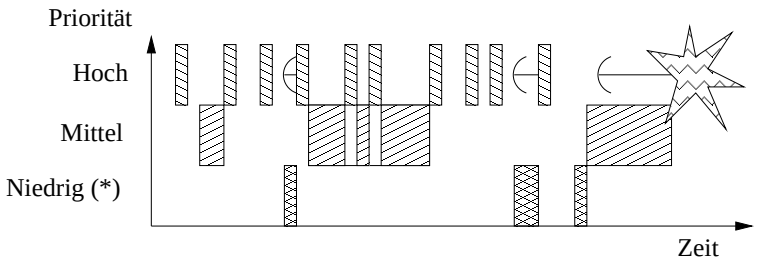
Dieses Problem ist in inhärent in vielen Systemen, in denen die Synchronisation auf Locks beruht.

Unter einem "Lock" soll hier im folgenden ein Mechanismus verstanden werden, mit dem ein Prozeß auf unbestimmte Zeit aufgehalten wird, bis ein anderer Prozeß "das Lock freigibt" (Mutexes, Semaphore und Monitore sind also Locks). In derartigen Systemen gibt es immer eine Möglichkeit, ein Deadlock zu konstruieren. Wir werden sehen, dass dies mit STM nicht mehr möglich ist.

Beobachtung Programme mit Locks sind nicht beliebig komponierbar. Zwei korrekte Teile müssen nicht ein korrektes Ganzes ergeben. Beispiel nach [HHMPJ05]:

Ein Wert soll von einer Hashtable in eine andere transportiert werden (aus A entfernt, in B eingefügt). Ein Zwischenzustand (zwei Einträge, kein Eintrag) soll für andere Threads nicht sichtbar sein.

Abbildung 1: grafische Darstellung einer Prioritätsinversion



In diesem Szenario könnten maximal beide Tabellen zum Lesen gesperrt werden, während der Wert transportiert wird. Dies würde sich jedoch nicht mit dem erwarteten Verhalten von Hashtables decken, und könnte bei einer existierenden Implementierung nicht ergänzt werden, wenn diese keine derartigen Locks bereitstellt.

1.3 Grundideen

Die aus der Datenbanktechnik bekannte transaktionale Semantik wird auf der Ebene der Hardware, des Betriebssystems oder der Laufzeitumgebung für den Hauptspeicher implementiert. Entwurfsziel ist "ACI":

- Atomicity - eine Transaktion ist entweder ganz erfolgreich oder gar nicht. Wenn sie nicht erfolgreich ist, wird sie wiederholt (roll back and retry), ohne dass ihre Schreibzugriffe wirksam werden.
- Consistency - nach jeder Transaktion verbleibt das System in einem definierten Zustand. Bei STM werden zur Zeit in keinem System explizite Konsistenzprüfungen unterstützt, allerdings muß die Umgebung immer noch garantieren, dass die die Verwaltung des Speichers wichtigen Daten ebenfalls konsistent bleiben.
- Isolation - nebenläufige Prozesse beeinflussen sich nicht gegenseitig.
- Eine über das übliche Maß hinausgehende Dauerhaftigkeit der Speicherung wird nicht verlangt.

Ausgezeichnete Speicherbereiche werden bei STM durch sogenannte Transaction Logs überwacht. Für das Verhalten der Umgebung zu diesen Speicherbereichen werden Garantien gegeben:

- Zugriff auf STM-überwachten Speicher ist nur mittels Transaktionen möglich.
- Schreibzugriffe bleiben für andere Threads unsichtbar bis zum Transaktionsende ("Commit").
- Lesezugriffe und Commits sind zueinander atomar, aber *kurz* (z.B.: Arbeitszeit linear zur Zahl der verwendeten Objekte).
- Wenn eine Transaktion von inzwischen geänderten Werten abhängt, wird sie spätestens beim Commit wiederholt.

Beobachtung: In jedem Konfliktfall kann eine Operation (die erste) ihren Commit vollenden, so dass jedes Mal eine Operation aus dem Pool der noch auszuführenden entfernt wird. STM ist also *lock-free*.

Lock-free bedeutet überigens nicht, dass man keine Fehler mehr machen kann: Wenn man z.B. auf eine Bedingung wartet, die niemals eintritt, wartet das Programm natürlich ewig. Lock-free bedeutet "nur", dass man kein Deadlock mehr aufbauen kann.

1.4 Transaktionslogs

Ein Transaktionslog speichert im Wesentlichen einen Zeiger auf eine Variable ($\pi(a)$), den ursprünglichen Wert von a (v) und eine Kopie von a (v'). Schreibzugriffe werden auf v' umgelenkt.

Am Ende der Transkation wird zuerst a mit v verglichen. Bei einer Unstimmigkeit wird die Transaktion wiederholt (die Transaktion hat einen

Wert gelesen, der danach geändert wurde). Stimmen a und v jedoch überein, wird die Differenz zwischen v und v' nach a geschrieben.

Wohlgemerkt darf zwischen beiden Operationen kein anderer Commit erfolgen. Es sind also strenggenommen immer noch Locks nötig, nur dass diese jetzt ausserhalb des Zugriffs des Programmierers liegen, und nicht dazu verwendet werden können, ein Deadlock zu konstruieren.

Verfeinert werden kann das Konzept durch Warteschlangen. Die Idee ist, dass eine Transaktion, die wiederholt wird, dies wohl wegen der transaktional gelesenen Werte getan hat[1]. Also wird sie in ein Warteschlange eingereiht und erst wiederholt, sobald sich einer der gelesenen Werte geändert hat (vgl. Monitore).

1.5 Implementation

Dieser Abschnitt fasst einige Erkenntnisse über die Integration von STM-Konzepten in Aspekte eines Computersystems auf.

Laufzeitumgebung verfügt über viel Information über die Software, bzw. könnte vom Compiler mit Zusatzinformation versorgt werden. Der Vorteil dieses Ansatzes liegt in der leichten Zugänglichkeit, und der möglichen Abstraktion in Richtung des Programms. Nötige Änderungen des Unterbaus (etwa: anpassung an eine andere Systemarchitektur) könnten vorgenommen werden, ohne dass bestehende Programme geändert werden müßten. Weiter unten werden verschiedene Ansätze für die Java VM und das GHC RTS im Detail besprochen.

Betriebssystem verfügt über viel Information über die Maschine, aber über wenig Information über die Software. Diese müßte dann irgendwie ergänzt werden (libraries, virtual machine...). "Rollback" auf Betriebssystemebene wird ohne Hardwareunterstützung ebenfalls ein schwieriges Unterfangen.

Hardware Auf Uniprozessorsystemen kann STM ohne weiteres realisiert werden, die einzige Schwierigkeit wäre die Integration von Interrupts. Teile der STM-Semantik könnten in Hardware realisiert werden. Beispielsweise einen existieren Maschinenbefehle, aus denen brauchbare STM-Primitive zusammengesetzt werden können. Zwei übliche Arten von Assemblerprimitiven werden hier erklärt.

compare and swap CAS verlangt drei Parameter: A, C und M (A und C sind typischerweise Register, M ist eine Hauptspeicher-Zelle). Falls $M = C$, werden M und A ausgetauscht. Ansonsten wird kein Wert verändert, aber ein Flag wird gesetzt. Vergleiche und Schreiboperationen sind zueinander atomar.

Bei Systemen, die ohnehin fast nur auf Zeigern arbeiten (wie z.B. die JVM, oder das GHC-RTS), ist CAS ausreichend, um STM zu implementieren, da ohnehin nur ein Zeiger geändert werden muß. CAS kann verallgemeinert werden zu einer ähnlichen Operation auf mehreren Prozessorwörtern [HFP02].

Auf IA32-Prozessoren heißt die Instruktion `cmpxchg`.

[1] anderenfalls hätte die Prüfung auf die Abbruchbedingung auch vorher erfolgen können – *werttreue*

load linked/store conditional LL ist ein lesender Speicherzugriff, der aber die Speicherzelle markiert. Wenn SC schreibend auf dieselbe Speicherzelle zugreift, wird überprüft, ob sich der Inhalt geändert hat. Geschrieben wird nur auf ungeänderte Inhalte. Der Unterschied zwischen CAS und LL/SC besteht also im wesentlichen darin, dass bei LL/SC zwischen Lese- und Schreiboperation einige Zeit vergehen darf, die genutzt werden kann, um Bedingungen an das gelesene Wort zu überprüfen.

Es ist offensichtlich, dass CAS trivial durch LL/SC implementiert werden kann. CAS ist schon hinreichend stark ist für viele blockadefreie Algorithmen [Gao05], auch wenn Algorithmen mit CAS üblicherweise komplizierter werden als dieselben Algorithmen mit LL/SC.

Weiterhin fällt auf, dass die Strukturen eines Cache eine gute Grundlage bilden, STM-Primitive in Hardware abzubilden. [HM93] Der Cache kann die Daten halten, bis die Transaktion abgeschlossen ist, da andere Threads ohnehin nicht darauf zugreifen können. Eine weitere Vereinfachung kann sich daraus ergeben, dass STM zuerst nur die Information braucht, *dass* sich Daten geändert haben. Die neuen Daten würden dann beim Wiederholen der Transaktion explizit gelesen.

> The idea is that the transactional cache holds all thetentative writes, without propagating them to other processors or to main memory unless the transaction commits. If the transaction aborts, the lines holding tentative writes are dropped (invalidated); if the transaction commits, the lines may then be snooped by other processors, written back tomemory upon replacement, etc. We assume that since the transactional cache is small and fully associative it is practical to use parallel logic to handle abort or commit in a single cache cycle. [HM93]

1.6 Skiplists

Traditionell werden sortierte Daten in einem Rot-Schwarz-Baum mit $O(log\, n)$ als Zeitschranke für die Algorithmen für Suche, Erweiterung und Löschung gespeichert. In einer Umgebung mit STM stellen sich solcherart balancierte Bäume als unpraktisch heraus. Gründe hierfür sind:

- Globale Invarianten müssen bewahrt werden. (STM zielt auf lokalisierung)
- Manche Algorithmen adaptieren schlecht auf STM (z.B.: wenn in einem Rot-Schwarz-Baum zur Vereinfachung der Implementation ein "sentinel"-Knoten benutzt wird, muß jeder Prozeß auf diesen Knoten warten)

Skiplists [Fra04] funktionieren ähnlich wie Listen mit "überbau". p-Skiplists der *Höhe* 1 sind gewöhnliche verkettete Listen. In Skiplists der *Höhe* 1 ist jedes Element mit der Wahrscheinlichkeit p *erwählt*. Eine p-Skiplist der *Höhe* $k+1$ wird aus einer der *Höhe* k erzeugt, indem jedes *erwählte* Element einen Zeiger zum nächsten erwählten erhält, und jedes *erwählte* Element mit einer Wahrscheinlichkeit von p *wiedererwählt* wird. (Siehe Abb. 2)

Es zeigt sich, dass Algorithmen existieren, Elemente zu suchen oder einzufügen, die auf die Referenzen sequentiell zugreifen, und die trotzdem

Abbildung 2: Eine 0.5-Skiplist der Höhe 3

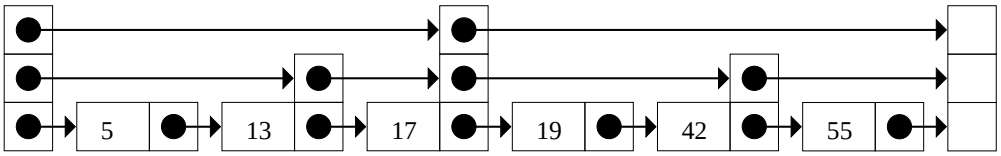


Abbildung 3: Entwurfsvorschlag für STM in Java [HLMS03]

```
Counter counter   = new Counter(0);
TMObject tmObject = new TMObject(counter);
```

`tmObject` überwacht jetzt `counter`

```
 // Hier wird die Kopie angelegt
Counter counterc = (Counter)tmObject.open(WRITE);
counterc.inc(); // Zähler inkrementieren

if (tmObject.close()) { // commit-versuch
   System.println("konnte nicht commiten")
}
```

in einem etwas abgewandelten Sinne korrekte Resultate liefern: Die Suche gibt ein Element zurück, das während *eines Zeitpunktes* während der Suche das Richtige war.

Anders als bei Rot-Schwarz-Bäumen wird hier keine strikte Laufzeitgarantie gegeben, sondern "nur" stochastische Aussagen. Die einfachere Implementierung und die Unwahrscheinlichkeit der schlechten Fälle jedoch lassen Skiplists als die insgesamt bessere Alternative erscheinen.

2 STM und Java

In diesem Abschnitt werden zwei Methoden erläutert, java durch STM-Semantik zu erweitern.

2.1 Java naiv

Ein Rahmenobjekt wird zur Verfügung gestellt, dass über den Konstruktor eine Referenz zum zu überwachenden Objekt erhält. Hierzu muß das überwachte Objekt ein `interface` implementieren

Die Transaktion wird diesem Objekt bekanntgegeben. Es legt eine Kopie an, gibt diese zurück, und läßt den Thread darauf zugreifen. Beim commit wird diese Kopie mit der überwachten verglichen.

Probleme

- Transaktion kann "offen" bleiben (Programmierfehler: `close()` vergessen - evtl. speicherleck).

Abbildung 4: Lösungsvorschlag zum Problem "`close` vergessen"

```
interface Transaction<A,B> {
  A transaction(B);
}
...
class TMObject<A> {
  ...
  private boolean open();
  private boolean close();
  ...
  public A transaction(
   Transaction <A,B> tr,B
   ) throws STMException
   {
     A a;

     if (!open()) throw ...;
     a=tr.transaction(B);
     if (!close()) throw ...;
     return a;
   }
}
...
tmObject.transaction(
 Transaction<counter,void> {
   return counter.inc(); });
```

Lösungsvorschlag: Das Typsystem von java kann mittlerweile zu recht mächtigen Konstruktionen verwendet werden. In der Abbildung 2.1 ist eine Konstruktion wiedergegeben, die verhindert, dass man eine Transaktion unvollständig läßt.

- Die Formulierung kann nicht sicherstellen, dass das ursprüngliche Objekt (hier `counter` nicht-transaktional geändert bzw. gelesen wird. Dass dies möglich ist, würde aber vermutlich erwartet.

  Lösungsvorschlag: im Moment der Konstruktion des Kapselobjekts (`TMObject`) wird schon eine private Kopie angelegt.

Es zeigt sich, dass zumindest eine brauchbare Annäherung an STM schon mit Java-Primitiven erlangt werden kann.

2.2 Java raffiniert

Einen etwas weiter gehenden Ansatz verfolgt [CCC+05][2]. Das Ziel dieser Entwicklung war, existierende Programme durch STM-Semantik zu "würzen", ohne dass die Programme zwangsläufig umgeschrieben werden müssen.

Hierbei wurden die vorhandenen Konstrukte `synchronized`, `volatile` und `Object.wait` verwendet und bekommen andere Funktionen. Während der Laufzeit wird anhand dieser Konstrukte entschieden, wo Transaktionen beginnen und aufhören.

- `Object.wait` bewirkt einen "Rollback".
- Nach jedem statement (an jedem `;` und an jedem codeblockende) muß ein Commit erfolgen, da jedes Statement überwachte Zustände ändern kann.
- `synchronized` unterbindet die Commits in einem Codeblock und in den Aufrufen, die darin geschehen. Es wird also eine große Transaktion über den gesamten synchronize-Block erzeugt.

[2]Folien bei `http://hdl.handle.net/1802/2096`

- Statements in Bezug auf `volatile`-referenzen sind ebenfalls Transaktionen. Man bemerke, dass volatile vorher schon eine STM-ähnliche Bedeutung besaß.

Es wurden also bestehende Metaphern zu diesen Keywords ausgenutzt. Exceptions wurden nicht eingeschränkt. Benchmark-Programme zeigten ähnliche Leistung, ob sie mit dieser Methode erweitert waren oder nicht.

Es ist zu erwarten, dass auf diese Entwicklung hin Implementierte Algorithmen vollen nutzen von dieser Einrichtung machen können.

3 Haskell

Die Funktionsweise von Haskell-Typklassen wird erläutert. Eine kurze Einführung in die Typklasse `Monad` wird gegeben, danach wird diese in den Zusammenhang mit IO bzw. konkurrenten Programmiertechniken gestellt.

3.1 Typklassen

Zuerst muß auf drei Probleme im Verständnis der Haskell-Typklassen hingewiesen werden:

- Typen beschreiben Daten, Typklassen die mit Typen mögliche Funktionen (reservieren Symbole und ordnen sie über das Typsystem dem richtigen Typen zu)
- *Typklassen sind Typconstraints* Eine Typklasse kann nicht "instanziiert" werden wie eine `class` in java. (Keine Speicherfunktion!)
- `class` definiert die Symbole einer Typklasse und deren Typen. `instance Klasse Typ` definiert den Körper dieser Funktionen einer Klasse für einen Typ.

Typklassen in Haskell sind eine Detaillierung der Möglichkeiten eines Typs.

Die Sichtweise sei der in der OOP üblichen entgegengesetzt:

java	haskell
Alle Objekte erben von `Object`. `Object` ist somit der allgemeinste Constraint.	Ein Objekt ohne Constraint ist nicht beschränkt
Erweiterung kann den Speichertyp ändern	Keine Veränderung des Speichertyps
Erweiterung der Funktionalität	Erklärung implizierter Funktionalität (generische Instanziierung)
Fast alles wird in Methoden erledigt	Instanzierung hat eng umrissenes Aufgabenfeld

Beispiel

$$\text{fmap} :: (\text{Functor } f) \Rightarrow (a \to b) \to f\ a \to f\ b$$

Diese Verwendung von Typklassen ist die für uns wichtige. Sie schränkt hier im Beispiel das Objekt f auf Objekte mit der Eigenschaft "Funktor" ein. Man beachte, dass f einen Parameter besitzt (Kind $* \to *$).

Beispiel in ghci, tippe:

```
Prelude> :m Data.Char
Prelude Data.Char> fmap (chr.(+10).ord) ['a'..'i']
"klmnopqrs"
Prelude Data.Char> :t chr.(+10).ord -- Typabfrage
chr.(+10).ord :: Char -> Char
Prelude Data.Char> :t ['a'..'i']
['a'..'i'] :: [Char]
Prelude Data.Char> :t fmap
fmap :: (Functor f) => (a -> b) -> f a -> f b
Prelude Data.Char> :t fmap (chr.(+10).ord)
fmap (chr.(+10).ord) :: (Functor f) => f Char -> f Char
Prelude Data.Char> :t fmap (chr.(+10).ord) ['a'..'i']
fmap (chr.(+10).ord) ['a'..'i'] :: [Char]  – (f = [])
```

Hier wurde also `[]` für f eingesetzt: `[]` ist ein Funktor

`Monad` ist eine besondere dieser *Typklassen*, also ein "constraint" über Typen. Typconstraints in haskell beinhalten Beschränkungen der *kind*[3]. `Monad` betrifft - ähnlich wie das im Beispiel erwähnte `Functor` Typausdrücke mit *kind* $* \to *$.

Kinds (Beispiele)

$*$	$* \to *$	$* \to * \to *$
Int		
String		
$(a, b) = (,)\ a\ b$	$(,)\ a$	$(,)$
$(a, b, c) = (,,)\ a\ b\ c$	$(,,)\ a\ b$	$(,,)\ a$
$[a] = []\ a$	[]	
Maybe a	Maybe	

3.2 Die Klasse Monad

Hat die zwei Funktionen

$$\begin{aligned} \text{return} &:: a \to m\ a \\ \gg\!= &:: m\ a \to (a \to m\ b) \to m\ b \end{aligned}$$

Wie man leicht sieht, ist mit $\gg\!=$ ein verketten von voneinander abhängigen Werten auf eine bestimmte Weise möglich. Zusätzlich werde von einer Monade gefordert, dass sie gewisse Gesetze[4] erfüllt:

$$\begin{aligned} (return x) \gg\!= f &= f\ x \\ m \gg\!= return &= m \\ (m \gg\!= f) \gg\!= g &= m \gg\!= (\lambda x \to f\ x \gg\!= g) \end{aligned}$$

Es zeigt sich, dass diese Verkettung ausreicht, das Problem des Werttreue/IO-Konflikts in funktionalen Sprachen zu lösen. Zudem geschieht das auf

[3] Metatyp, Typ des Typs. In haskell wird nur dessen Arität gezählt

[4] die nicht von ungefähr den Axiomen für ein *Mon*oid ähneln

transparente Weise: das Typsystem transportiert Information darüber, wo Wirkungen (bzw. fehlende Werttreue) zu erwarten ist.

3.2.1 Arbeitsmetapher für Monaden

Da der Begriff "Monade" oft einschüchternd wirkt, sei hier eine Metapher erwähnt, die dem Verständnis helfen könnte.

Eine "Monade" (der Typ von Kind $* \to *$ ist gemeint) ist ein "Skripttyp", ein "Monadischer Wert vom Typ m a" (mit einer Monade m) dagegen ein "Skript vom Typ m mit einem Rückgabewert von Typ a".

Beispiel:

- getLine :: IO String — Skript vom Typ IO mit Rückgabewert String
- putStr :: String → IO () — Funktion mit einem Parameter vom Typ String. Für einen Eingabeparameter wird ein Skript zurückgegeben, das seinerseits keinen Rückgabewert hat. Der Parameter darf auch "rein" berechnet werden.

Ein paar für den Umgang mit Monaden wichtige Funktionen sollen hier nicht unerwähnt bleiben:

- sequence :: (Monad $m \Rightarrow [m\,a] \to m\,[a]$ erlaubt die serielle ausführung einer Liste von Skripts. Wird das Ergebnis sowieso verworfen, so kann sequence_ verwendet werden, das dann keinen Speicher mehr leckt.
- mapM :: Monad $m \Rightarrow (a \to m\,b) \to [a] \to m\,[b]$ map in einer Monade.

Gelegentlich spielt auch `MonadPlus` eine Rolle, das eine zweite Operation (`mplus`) unterstützt, die ein neutrales Element besitzt (`mzero`) und mit der ersten kommutiert.

- mplus :: MonadPlus $m \Rightarrow m\,a \to m\,a \to m\,a$
- mzero :: MonadPlus $m \Rightarrow m\,a$
- msum :: MonadPlus $m \Rightarrow [m\,a] \to m\,a$ verallgemeinerung von mplus.

3.2.2 Die Maybe-Monade

$$\text{data Maybe } a = \text{Nothing} | \text{Just } a$$

Kann eine Monade sein, und dann Berechnungen repräsentieren, die vielleicht kein Ergebnis liefern. $\gg=$ hat zwei Fälle zu beachten:

instance Monad Maybe where
$\quad$ Just $x \quad \gg= k = k\ x$
$\quad$ Nothing $\gg= k$ = Nothing

Maybe ist eine `MonadPlus`, `Nothing` ist das neutrale Element von `mplus`

3.2.3 Die Listenmonade

Ein interessanter Spezialfall ist die Listenmonade. Sie entspricht durch die Bedarfsauswertung in Haskell einem Backtracking-Algorithmus (*!*).

```
filter even ([2,3,4] >>= \x->[x*2,x*3] >>= \x->[x+1,x+2])
[6,8,8,10,10,14]
```

In der Listenmonade entspricht $\gg=$ der Operation

$$\lambda\, x\, y \rightarrow \mathrm{concat}(\mathrm{map}\, y\, x)$$

also:

```
[2,3,4] >>= \x->[x*2,x*3] >>= \x->[x+1,x+2]
concat (map \x->[x*2,x*3] [2,3,4]) >>= \x->[x+1,x+2]
concat ([[4,6],[6,9],[8,12]]) >>= \x->[x+1,x+2]
[4,6,6,9,8,12] >>= \x->[x+1,x+2]
concat (map \x->[x+1,x+2] [4,6,6,9,8,12])
concat ([[5,6],[7,8],[7,8],[10,11],[9,10],[13,14]])
[5,6,7,8,7,8,10,11,9,10,13,14]
```

wenn jetzt nur ein Teil der Ergebnisse gebraucht wird, oder eines, das eine bestimmte Eigenschaft erfüllt, werden nur soviele Listenelemente wie nötig[5] ausgewertet:

```
> head (filter (300<) ([1..] >>= \x->[x*17,x*19,x*23,x*29]))
319
```

Die Liste ist ebenfalls eine Instanz von MonadPlus, mit [] als neutralem Element. `mplus` entspricht einfach der Konkatenation, stellt im Sinne des Backtracking also eine Alternative dar.

3.2.4 ST-Variablen

Über die Monade `ST` sind mit `STRef` *muteable* Referenzen verfügbar (Data.STRef.newSTRef, readSTRef, writeSTRef, modifySTRef). Anders als IO-Skripte ist ST jedoch werttreu und kann somit aufgelöst werden:

$$\mathrm{runST} :: (\forall s.\mathrm{ST}\ s\ a) \rightarrow a$$

Der genaue Typ des Zustandes (s) ist Opak und wird vom Compiler erzeugt. Innerhalb dieser Monade kann jetzt imperativ programmiert werden.

3.2.5 Schreibabkürzungen

```
a>>=b>>=c
```

ist dasselbe wie

```
do a
   b
   c
```

```
a>>=\x->b>>=\y->c y
```

ist dasselbe wie

```
do x<-a
   y<-b x
   c y
```

3.3 Zusammenhang zu IO und konkurrenter Programmierung

Die `main`-Funktion eines Haskell-Programms erhält nun den Typ `IO ()`[6] , stellt also ein IO-Skript dar. Es kann im wesentlichen:

[5] die Liste im Beispiel hat unendlich viele Elemente

[6] Zur vereinfachung wird bei anderen Typen ein `return` ergänzt um den Rückgabetyp in die Monade zu *heben*.

- andere IO-Skripts aufrufen
- werttreue Berechnungen ausführen, mit Parametern, die ggf. von IO abhängen.

Andererseits ist jetzt von innerhalb werttreuer Berechnungen kein IO mehr möglich (haskell ist werttreu).

Die "Kapselung" von IO-Skripts legt nahe, Thread-Semantik ebenfalls "innerhalb" dieser Monade zu organisieren. Und in der Tat findet sich:

```
forkIO :: IO () -> IO ThreadId
```

welches ein IO-Skript parallel ablaufen läßt. Zur Kommunikation gibt es außer den IORefs noch `Control.Concurrent.MVar` (Synchronisierte Variablen), und `...Chan`, das "message Passing" zur Verfügung stellt.

4 STM in Haskell

[HHMPJ05]

Die STM-Operationen sind im wesentlichen gleichwertig zu IO-Operationen (Operationen mit Wirkung auf die Umgebung). Um den Anteil an STM-überwachtem Code möglichst gering zu halten, werden STM-Operationen werden in einer eigenen Monade dargestellt. Durch diese einfache Maßnahme werden die technisch verwandten Operationen für IO und STM voneinander getrennt und unterscheidbar gemacht. Ein STM-Skript ist jetzt also beschränkt auf STM-Operationen und reine Berechnungen.

Kombinatoren

- $\gg\!=$ :: STM $a \rightarrow (b \rightarrow \text{STM } b) \rightarrow \text{STM } b$

  Sequenz: wenn die erste Transaktion fehlschlägt, schlägt das Ergebnis auch fehl. Wenn sie nicht fehlschlägt, bekommt der zweite Parameter das Ergebnis zur Verfügung gestellt.
- orElse :: STM $a \rightarrow$ STM $a \rightarrow$ STM a

  Implementiert eine Alternative: wenn die erste Operation fehlschlägt, wird die Zweite ausgewertet, statt die erste zu wiederholen. Wenn diese auch fehlschlägt, schlägt die Kombination fehl.
- retry :: STM a läßt die Transaktion wiederholen.

Innerhalb einer STM-Transaktion wird also ein Entscheidungsbaum mittels der Kombinatoren aufgebaut. Ähnliche Konstruktionen sind aus der imperativen Programmierung hinlänglich bekannt[7].

(STM a,orElse,retry) bilden ein Monoid:

M1 'orElse' (M2 'orElse' M3)	=	(M1 'orElse' M2) 'orElse' M3
retry 'orElse' M	=	M
M 'orElse' retry	=	M

$\Rightarrow$ STM ist eine MonadPlus.

Als letztes wichtiges Konstrukt bleibt `atomically`, das einen Typ von STM$a \rightarrow$ IOa hat, also ein STM-Skript in ein IO-Skript übersetzt. Man beachte, dass man STM-Skripte ohne `atomically` überhaupt nicht ausführen kann.

[7] Shell-Scripts, perl, C

Mit Hilfe der bis jetzt besprochenen Konstrukte können also STM-Skripts beliebig kombiniert werden, mit reinen Berechnungen vervollständigt werden und schließlich ausgeführt werden.

4.1 TVar

Ähnlich die schreibbaren Variablen in IO und ST können mit TVar schreibbare Variablen verwendet werden. Auf diese kann natürlich nur über STM-Operationen zugegriffen werden:

- newTVar:: $a \rightarrow$ STM(TVar$a$)
- readTVar:: TVar$a \rightarrow$ STMa
- writeTVar:: TVar$a \rightarrow a \rightarrow$ STMa

Es wird also sichergestellt, dass auf STM-Speicher nur innerhalb von STM zugegriffen wird, welches nur innerhalb von atomically verwendet werden kann, welches schließlich die Umsetzung in tatsächliche Speicheroperationen bewirkt.

4.2 Multicast-Kanäle per TVar

Ein Multicast-Kanal ist ein Kanal mit einem Sender und mehreren Empfängern, so dass jedes gesendete Signal von jedem registrierten Empfänger empfangen wird. [HHMPJ05] demonstriert eindrucksvoll, wie Multicast-Kanäle über STM implementiert werden können.

```
type Chain a = TVar (Item a)
data Item a = Empty | Full a (Chain a)

type MChan a = TVar (Chain a)
type Port a = TVar (Chain a)
```

Sowohl der Kanal als auch der Leseport werden durch eine Referenz auf eine `Chain` implementiert (also eine Referenz "zweiten Grades" auf ein `Item`, hierbei zeigt `MChan` stets auf das Schreibende der Warteschlange und `Port` auf eine Stelle, an der zu lesen ist.

```
newMChan :: STM (MChan a)
newMChan = do c ← newTVar Empty
              newTVar c
```

Einen leeren Kanal neu zu erzeugen ist einfach: eine Referenz auf eine Referenz, die auf einen Wert `Empty` verweist.

```
newPort :: MChan a → STM (Port a)
newPort mc = do c ← readTVar mc
                newTVar c
```

Für einen Port wird einfach der Zeiger auf das Schreibende kopiert. Der Port liest dann alle Daten, die von diesem Zeitpunkt an in den Channel geschrieben werden.

```
readPort :: Port a → STM a
readPort p = do c ← readTVar p
                i ← readTVar c
```

```
case i of
        Empty → retry
        Full v c' → do writeTVar p c'
                       return v
```

Um von einem Port zu lesen, werden beide Referenzen aufgelöst. `Empty` bedeutet, dass der Sender nichts mehr geschickt hat, und führt zu einer Wiederholung.

```
writeMChan :: MChan a → a → STM ()
writeMChan mc v = do c ← readTVar mc
                     c' ← newTVar Empty
                     writeTVar c (Full v c')
                     writeTVar mc c'
```

Und um in einen Kanal zu schreiben, wird einfach das `Empty` durch einen passenden `Full`-wert ersetzt.

4.3 Problem der dinierenden Philosophen

Beispielhaft werden hier Funktion aus einer Implementation der dinierenden Philosophen wiedergegeben und diskutiert.

Occupy Diese Funktion belegt eine einzelne Gabel. Dabei wird eine Identifikationsnummer hinterlegt. Dieses Skript wiederholt, wenn die Gabel belegt ist.

```
type Fork = TVar (Maybe Int)

occupy :: Int → Fork → STM ()
occupy i f = do x ← readTVar f
                if (x ==Nothing)
                    then writeTVar f (Just i)
                    else retry
```

Eat Diese Funktion läßt einen durch eine Zahl identifizierten Esser zwei Gabeln greifen.

```
eat :: Int → Fork → Fork → IO String
eat i a b = atomically ((do occupy i a
                            occupy i b
                            return "Nudeln"))
—                       'orElse'
—                       (error "Konnte nicht Gabeln." ))
```

Da die Darstellung von Nudeln uninteressant ist, wird ein String zurückgegeben. Sollte eine der `occupy`-Funktionen abbrechen (`retry`), wird der ganze Block warten gelassen, bis sich einer der gelesenen Werte (`TVar` in `Fork`) ändert. Man beachte, dass in diesem Fall eine andere Transaktion

commited wurde. Es gibt also mindestens einen Thread, der Fortschritt macht.

Die auskommentierte Variante dient nur der Demonstration von `orElse`: `error` ist Totalabbruch des Programms. Ohne `orElse` wird die Transaktion dann wiederholt.

alieneat Natürlich können jetzt Funktionen höherer Ordnung eingesetzt werden um zu abstrahieren. Hier z.B. werden beliebig viele Gabeln atomar gegriffen:

```
alieneat :: Int → [Fork] → IO Int
alieneat id fl = atomically (do sequence_ (map (occupy id) fl)
                                return (length fl)) – "reine" berechnung
```

5 Diskussion

5.1 Eingliederung

STM weist folgende Merkmale auf:

- expliziter Parallelismus, explizite Dekomposition
- implizite Synchronisation und Kommunikation

Beachtenswert ist, dass das Typsystem von Haskell ausreicht, um sinnvolle Einschränkungen durchzusetzen.

5.2 Fazit

Das STM-Programmiermodell bietet eine einfach verständliche Abstraktion von parallelen Prozessen und läßt sich in vielen Umgebungen umsetzen. Bei naiver Umsetzung von nicht-STM-Algorithmen wird in der Regel kein Schaden angerichtet [CCC+05]. Für Gewinn bei der Performance müssen eventuell andere Datenstrukturen verwendet werden. Viele derartige Strukturen und die zugehörigen Algorithmen sind bekannt und untersucht.

Literatur

[CCC+05] Brian D. Carlstrom, JaeWoong Chung, Hassan Chafi, Austen McDonald, Chi Cao Minh, Lance Hammond, Christos Kozyrakis, , and Kunle Olukotun. Transactional execution of java programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. ?, Oct 2005.

[Fra04] Keir Fraser. Practical lock-freedom. University of Cambridge, Technical Report UCAM-CL-TR-579/Ph.D. Thesis, 2004.

[Gao05] Hui Gao. Design and verification of lock-free algorithms. book on demand: ISBN 90-367-2231-4, http://irs.ub.rug.nl/ppn/274119137, 2005.

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.

[HHMPJ05] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton-Jones. Composable memory transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, to appear*, Jun 2005.

[HLMS03] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2003.

[HM93] Maurice Herlihy and Eliot Moss. Transactional memory: Architectural support for lock-free data structures. *Proceedings of the Twentieth Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[Jon97] Mike Jones. `http://research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html`, 1997.