Haskell Transaktionen in verteilten Systemen

Frank Huch und Frank Kupke

Christian-Albrechts-Universität zu Kiel, Institut für Informatik
Olshausenstr. 40, 24098 Kiel, Germany
{fhu,frk}@informatik.uni-kiel.de

Abstract. Das Konstrukt der Transaktions-Variablen in Concurrent Haskell erleichtert die Entwicklung von nebenläufigen Anwendungen, indem es Deadlocks verhindert und erlaubt Programmteile einfach zu komponieren. In einer vorherigen Arbeit haben wir bereits eine Haskell-Version als Alternative zur existierenden lowlevel GHC-Bibliothek implementiert, die das optimistische verify-und-commit Konzept durch das Sammeln von monadischen Aktionen realisiert. Dieses Prinzip haben wir auf verteilte Systeme übertragen. Wir zeigen zunächst eine erste nicht robuste Lösung, welche die zu Synchronisation und Nachrichtenaustausch benötigten polymorphen Transaktions-Variablen zur Überwindung der Prozessgrenzen in eine externe Variante wandelt. Als nächste Herausforderung untersuchen wir die Speicherverwaltung der verteilten Transaktions-Variablen und das Verhalten bei terminierenden Prozessen, um die Robustheit der Bibliothek zu erhöhen.

1 Transaktionen in Concurrent Haskell

Um nebenläufige Programmierung zu ermöglichen, muss die verwendete Programmiersprache die Erzeugung und Verwaltung von *Threads* ermöglichen sowie Synchronisierung und Kommunikation zwischen den Threads unterstützen. Concurrent Haskell bietet für letzteres sogenannte *veränderbare Variablen (mutable variables)* an. Veränderbare Variablen können immer nur von einem Thread zu einer Zeit bearbeitet werden. Andere Threads, die ebenfalls zugreifen wollen, blockieren, bis der erste Thread die Variable zurückgelegt hat. Dies führt sehr leicht zu *Deadlocks* und anderen unerwünschten Effekten.

Harris, Marlow, Peyton-Jones und Herlihy haben eine neue Abstraktion für Concurrent Haskell vorgeschlagen: *software transactional memory* (STM) [2]. Der Ansatz beruht auf dem Transaktionskonzept von Datenbanken und erlaubt dem Anwender *atomar* auszuführende Aktionen zu definieren, die entweder vollständig ausgeführt werden oder aber keinen Effekt haben. Die Aktionen werden solange erneut ausgeführt, bis alle Transaktionen erfolgreich beendet sind. Mit diesem optimistischen Ansatz lassen sich effektiv und für den Anwender transparent Deadlocks verhindern. Dieser Ansatz ist effizient mit externen C Primitiven im Glasgow Haskell Compiler, Version 6.4 (*ghc*) [1] implementiert. Wir haben eine Implementation in Concurrent Haskell vorgestellt[3], die hohe Portabilität bei geringem Effizienzverlust bietet.

Das Transaktionskonzept für Concurrent Haskell stellt für Kommunikation und Synchronisation der Threads *Transaktions-Variablen (transactional variables)* über folgendes Interface zur Verfügung:

```
-- Software Transactional Memory
data STM a  -- abstract
instance Monad STM

-- Running STM computations
atomically :: STM a -> IO a
retry :: STM a
orElse :: STM a -> STM a -> STM a

-- Transactional variables
data TVar a  -- abstract
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Die Kernidee des Transaktionskonzeptes ist das optimistische Auswerten der Einzelaktionen auf den `TVar`s im Schutz der STM-Monade. Am Ende der gesamten Transaktion wird die Validität aller verwendeten `TVar`s geprüft (*Validierungs*-Aktion). Bei positivem Ergebnis werden die Ergebnisse aus der Monade heraus bestätigt (*Commit*-Aktion). Validierungs- und Commit-Aktion laufen atomar ab. Bei negativem Ergebnis werden alle Einzelaktionen zurückgesetzt und die gesamte Transaktion erneut gestartet.

Wir haben die STM-Monade als Zustandsmonade modelliert, die während der Transaktion die einzelnen Validierungs- und Commit-Aktionen sammelt. Am Ende der Transaktion werden alle gesammelten Aktionen dann atomar ausgeführt.

TVars bestehen neben dem eigentlichen Inhalt (IORef für Zeigervergleich bei Validitätsprüfung) aus weiteren Elementen:

```
data TVar a = TVar (MVar (IORef a)) -- the TVar content
              TVarID                -- index
              (MVar ())             -- TVar lock
              (MVar [MVar ()])      -- wait queue on retry
```

Das *Lock* wird für die atomaren Zugriffe benötigt. Über die Warteschlangenverwaltung verhindert man "Busy Waiting" bei invaliden Transaktionen. Die STM-Monade wird erst dann neu gestartet, wenn mindestens eine der gelesenen TVars von anderen Transaktionen verändert wurde.

2 Transaktionen in verteilten Systemen

Threads in nebenläufigen Programmen laufen in einem Prozessknoten mit einer gemeinsamen Speicherverwaltung ab. In verteilten Systemen kann jeder Thread und auch jede TVar ein eigener Prozessknoten sein. Wir stellen eine Lösung vor, die das Haskell Transaktionskonzept auf verteilte Systeme überträgt und deren Semantik dabei möglichst erhält. Die Bibliothek muss allerdings ergänzt werden. Weiterhin soll sich das System weitgehend robust gegen Störungen verhalten.

In verteilten Systemen findet Synchronisation mit Hilfe von *IP*-Kommunikation statt. Die TVars müssen durch IP-Nachrichten zwischen den Prozessknoten bekannt gemacht werden. Zeiger und veränderbare Variablen können jedoch nicht sinnvoll versendet werden. Deshalb haben wir den Datentyp TVar entsprechend erweitert:

```
data TVar a = TVar (MVar (a, VersionID)) -- the TVar content
              TVarID                     -- index
              (MVar ())                  -- TVar lock
              (MVar [RetryVar])          -- wait queue on retry
            | RemTVar EnvAddr            -- IP-addr + port
                      TVarID             -- index

data RetryVar = RetryLocal (MVar ())   -- wait queue element
                           RetryVarID  -- index
              | RetryRemote EnvAddr    -- IP-addr + port
                            RetryVarID -- index
```

Die lokale Variante der TVar kann in die Remote-Variante umgewandelt und als IP-Nachricht an einen entfernten Prozess (Thread) verschickt werden. Der entfernte Prozess kann nun über Nachrichten mit dem Indexeintrag in Datenstrukturen des Original-Prozessknotens auf die Original-TVar zugreifen. Der Zeigervergleich bei der Validitätsprüfung wird durch einen Vergleich von Versionsnummern ersetzt. Die veränderlichen Variablen der Retry-Warteschlangen werden ebenfalls durch Remote-Varianten ersetzt.

Die Implementation der Zugriffsroutinen auf die TVars kann durch Haskell-Patternmatching weitgehend analog zu den nebenläufigen Varianten erfolgen. Bei einem Zugriff auf eine lokale TVar wird dieser direkt ausgeführt. Soll eine TVar gelesen oder geschrieben werden, die auf einem anderen Prozessknoten liegt, verschickt die Remote-Alternative der jeweiligen Routine eine Nachricht an den Prozess, in dem die Original-TVar liegt. Jeder Prozessknoten startet einen Empfänger-Thread, um jederzeit solche Anfragen ausführen zu können. Da zum Zeitpunkt der Ausführung die Original-TVar nicht mehr bekannt ist, hat der Prozess, in dem sie liegt, vorher alle notwendigen Zugriffe als IO-Aktionen gespeichert.

Für kleine Testapplikationen funktioniert das vorgestellte Verfahren gut. Wie verhalten sich aber große Applikationen mit vielen TVars, die sehr lange in Betrieb sind? Ist das System robust gegen Störungen? Wir geben eine erste Antwort auf mögliche Speicherprobleme mit Hilfe der Haskell *finalizer* und werden die Robustheit weiter untersuchen, um auch hier zu befriedigenden Lösungen zu gelangen.

References

1. The Glasgow Haskell compiler. http://www.haskell.org/ghc/.
2. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.
3. Frank Huch and Frank Kupke. Composable Memory Transactions in Concurrent Haskell. In A. Butterfield, editor, *Implementation and Application of Functional Languages, 17th INternational Workshop, IFL'05, Selected Papers*, volume ??? of *LNCS*. Springer, 2006. to appear.