Transactional Events: A Brief Overview

Christopher R. Wicks
9/19/2017

Introduction

Transactional events are a relatively recent (2006) concurrency abstraction in the functional programming paradigm. They are a high-level abstraction that combine the first-class synchronous message-passing events of Concurrent ML (CML) with all-or-nothing transactions.[1] Their inception draws inspiration from CML, Concurrent Haskell, and Software Transactional Memory (STM) Haskell. There exist implementations in both Haskell [1] and ML.[2]

Motivation

Transactional events are primarily motivated by the need for better abstractions in the development of concurrent programs. The non-deterministic execution order of concurrent programs make them inherently difficult to reason about. Transactional events help to ease some of the pain by allowing for inter-thread communications to be be composed of modular events in the form of transactions. Abstractly, a transaction groups several sequential actions that can either be committed or aborted based on some criteria. This is a well-known concept in the realm of databases or transactional memory. Transactional events extend this concept to synchronous message passing between threads using shared synchronous channels for inter-thread communications.

Haskell Interface and Usage

The basic interface for transactional events is given in [1], and is provided here for ease of reference.

```
data Evt a -- The Evt monad

sync :: Evt a -> IO a
thenEvt :: Evt a -> (a -> Evt b) -> Evt b
alwaysEvt :: a -> Evt a
chooseEvt :: Evt a -> Evt a -> Evt a
neverEvt :: Evt a

instance Monad Evt where
   (>>=) = thenEvt
   return = alwaysEvt

instance MonadPlus Evt where
    mplus = chooseEvt
    mzero = neverEvt

throwEvt :: Exception -> Evt a
catchEvt :: Evt a -> (Exception -> Evt a) -> Evt a

data SChan a -- Synchronous channels
newSChan :: Evt (SChan a)
sendEvt :: SChan a -> a -> Evt ()
recvEvt :: SChan a -> Evt a
```

One of the things that make transactional events a desirable tool for concurrent programming is the fact that they are easily composable due to their monadic structure. As seen here, in the Haskell implementation, the *Evt* typeclass is a subclass of both *Monad* and *MonadPlus*. This allows for easy sequencing of events utilizing Haskell's *do* notation. The monadic bind is implemented as *thenEvt*. An event composed (of other events) in sequence using the monadic bind can only successfully yield a result if and when all of the events of which it is composed successfully complete.

The interface also provides a non-deterministic choice combinator in the form of *chooseEvt*. This allows an event to synchronize on whichever of the two events it is able to synchronize on first, effectively aborting (from the view of the rest of the program) the computations performed in the event that is not chosen. Note that *neverEvt* is the event that can never be synchronized upon. Hence, sequencing a *neverEvt* using the monadic bind is an effective abort; likewise the *chooseEvt*, as it cannot choose an event that will not successfully complete, will never choose the *neverEvt*.

Using this interface, one can easily compose sophisticated computational event structures that allow for inter-thread communication but guarantee that irrevocable side effects can not take place during event synchronization (hence the "all-or-nothing" transactional claim). Indeed, in [1] the authors provide TE Haskell encodings for both STM Haskell and CML.

References

[1] Kevin Donnelly and Matthew Fluet. 2006. Transactional events. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming* (ICFP '06). ACM, New York, NY, USA, 124-135.

[2] Laura Effinger-Dean, Matthew Kehrt, and Dan Grossman. 2008. Transactional events for ML. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming* (ICFP '08). ACM, New York, NY, USA, 103-114.