Software-Transactional Memory in Haskell
(an overview of the implementation)

Let's start with WHY

FACT:
Many modern applications have increasingly stringent concurrency requirements

FACT:
Commodity multicore systems are increasingly affordable and available

FACT:
The design and implementation of correct, efficient, and scalable concurrent software remains a daunting task

Haskell to the rescue!

Meet STM

STM
protects shared state in concurrent programs

STM

provides a more user-friendly and scalable alternative to locks by promoting the notion of memory transactions as first-class citizens

Transactions, like many of the best ideas in computer science, originated in the data engineering world

Transactions
are one of the foundations of
database technology

Full-fledged transactions are defined by the ACID properties

Memory transactions use two of them (A+I)

Transactions provide atomicity and isolation guarantees

Strong atomicity means all-or-nothing

Strong isolation means freedom from interference by other threads

Recall that Haskell is a strictly-typed, lazy, pure functional language

Pure means that functions with side-effects must be marked as such

The marking is done through the type system at compile time

STM
is just another kind of
I/O
(with a different marker: "STM a" instead of "IO a")

Transactional memory needs to be declared explicitly as TVar

The STM library provides an STM-to-IO converter called "atomically"

Transactional memory can only be accessed through dedicated functions like "modifyTVar", "readTVar", "writeTVar" which can only be called inside STM blocks

Implementation Overview Of GHC's STM

Definition

A transaction memory is a set of tuples in the shape of (Identity,Version,Value)

The version number represents the number of times the value has changed.

The Transactional Record

Every STM transaction keeps a record of state changes

(similar to the tx log in the DB world)

STM performs
all the effects
of a transaction
locally
in the transactional record

Once the transaction
has finished its work locally,
a version-based
consistency check
determines
if the values read
for the entire access set
are consistent

This version-based
consistency check
also obtains locks
for the write set
and with those locks
STM updates
the main memory
and then releases the locks

Rolling back the effects of a transaction means forgetting the current transactional record and starting again

Reading: When a readTVar is attempted STM first searches the tr. record for an existing entry

Reading: If the entry is found, STM will use that local view of the TVar

Reading: On the first readTVar, a new entry is allocated and the TVar value is read and stored locally

Reading: The original Tvar
does not need
to be accessed again
for its value
until
validation time

Writing: **Writing to a Tvar requires that the variable first be in the tr. record**

Writing: If it is not currently in the tr. record, a readTVar is performed and the value is stored in a new entry

Writing: The version in this entry will be used at validation time to ensure that no updates were made concurrently to this TVar

Writing: The value
is stored locally
in the tr. record
until commit time

Validation: Before a transaction can make its effects visible to other threads it must check that it has seen a consistent view of memory while it was executing

Validation: This is done by checking that TVars hold their expected values (version comparison)

Validation: During validation, STM fetches the version numbers for all TVars and checks that they are consistent with its expectations

Validation: STM then acquires locks for the write set in ascending order of memory address

Validation: STM then reads and checks all version numbers again

Validation: If the version numbers are again consistent with its expectations, STM allows the commit to happen

Committing: The desired atomicity is guaranteed by:

- Validation having witnessed all TVars with their respective expected values
- Locks being held for all of the TVars in the write set

Committing: STM proceeds to increment each locked TVar's num_updates (a.k.a. version) field

Committing: STM then writes the new values into the respective current_value fields, and releases the locks

Committing: While these updates happen one-by-one, any attempt to read from this set will spin while the lock is held

Another useful STM abstraction is the TChan, an unbounded FIFO channel

Once some messages are transferred into a TChan, they are ready to be consumed by other threads (broadcasting is possible too)

TChans are useful when threads need to send signals to each other, as opposed to just accessing shared state

Compile your STM code with:

```
ghc -threaded program.hs
```

When running the program:

```
./program +RTS -N
```

Follow me on GitHub

github.com/dserban