Software Transactional Memory

Pablo Castro

McMaster University

Concurrent Haskell

Conc. Haskell is an extension of Haskell that allows it to express concurrent applications. It has the following characteristics:

- It uses only four new primitive operations.
- A primitive to create new threads (`fork`).
- A data structure to provide communication between threads.

Concurrency in Haskell

In this presentation we will see the following topics:

- Brief introduction to Concurrent Haskell
- Introduction to Transactional Memory
- Some Examples.
- Drawbacks and Benefits of STM.
- Conclusions

Concurrent Haskell: Primitives

- `forkIO()::IO() -> IO()`: spawns a new process.
- `type MVar a`: a mutable location, it allows the interaction between processes.
- `newMVAr::IO (MVar a)`: creates a new `MVAR`.
- `takeMVar::MVar a -> IO a`: blocks until the location is non-empty, then reads a value.
- `putMVar::MVar a -> a -> IO()`: writes a value in the specified location. It is an error if the location is full.

C.Haskell: Example 1

```
let loop ch =
hPutChar stdout ch >> loop ch
in
   forkIO(loop 'a') >> loop 'b'
```

question: What is the behaviour of this program?

Answer: Threads are interleaved in an unspecified way. The result is some infinite sequence of a's and b's.

C.Haskell: Example II

We can use `MVar` to model synchronic communication:

```
type CVar = (MVar a, MVar())

newCVar :: IO(CVar a)
newCVar = newMVar >>= \data_var ->
          newMVar >>= \ack_var  ->
          putMVar ack_var () >>
          return (data_var, ack_var)

putCVar :: CVar a -> a -> IO()
putCVar (data_var, ack_var) val =
                 takeMVar ack_var >>
                 putMVar data_var val
```


Communication between Threads

We need a mechanisms for communication and synchronisation, it is provided by `MVar`:

- `MVar t`, it can be seen as a reference to a value of type `t`.
- It can be used as a channel, where `takeMvar` play the role of send and `putMVal` plays the role of receive. (it is asynchronic)
- the type `MVar()` is a semaphore.

Example II...

```
getCVar :: CVar a -> IO a
getCVar (data_var, ack_var) =
         takeMVar data_var >>= \val ->
         putMVar ack_var ()
         return val
```

Software Transactional Memory

Usually, in concurrency, lock mechanisms are used to allow the interaction between process, But this methodology has some drawbacks.

- Hard to use and understand.
- Correctness is not easy to prove (for example, O'Wicki-Hoare logic)
- Components are not composable, in general.

STM

The introduction of STM in Haskell has the following characterics:

- A transation (block) is declarated using `atomic{...}`
- An `atomic` block runs without locking with a transation log.
- Before the execution the log is validated to ensure consistency.
- We can use monads to provide encapsulation, and separated transactions from the other world.

Solution?: Transactional Memory

Same idea that *transactions* in data bases:

- A sequence of commands (including updates).
- Enclosed in an atomic way.
- If some conflict is produced during the transaction it aborts.
- The before state is restored (rollback).

STM interface

```
data STM a
instance Monad STM

-- exceptions
throw :: Exception -> STM a
catch :: STM a -> (Exception -> STM a) -> STM a

-- running STM computations
atomic :: STM a -> IO a
retry :: STM a
orElse :: STM a -> STM a -> STM a
```

STM ideas

- We can use `do` notation with `STM`
- `atomic`: separates the transactional world from the `IO`.
- `retry`: aborts the transaction without effects, and restarts it from beginning.
- `s ‘orElse‘ t`: Allows us to combine transactions, if `s` retries then `t` runs, if both retry then the transaction retries.
- `catch`, `throw`: enable exception mechanisms in STM.

An Example

We can model a resource manager, which holds an integer-valued resource:

```
type resource = TVVar Int
putR :: resource -> Int -> STM ()
putR r i = do { v <- readTVar r;
                writeTVar r (v+1)
```

Now, for `getR` we can use retry as a waiting intruction

Communication in STM

Communication between threads is possible using `TVar`:

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

Each thread transaction log to record the tentative accesses to `TVars`

Example: GetR

```
getR :: Resource -> Int -. STM()
getR r i = do { v <- readTVar r;
                if (v < i) then retry
                else writeTVar r (v-i)

}
```

Note that `putR` does not have to give some signal to the consumer, only writing and element the consumer wakes up.

Composing Alternatives

With locking mechanisms is hard to compose components. Using `orElse` we can do it easily:

```
atomic (getR r1 3 'orElse' getR r2 7)
```

With `orElse` we can have a more abstract view of the composition. And a *n-ary choice* becomes possible:

```
merge :: [STM a] -> STM a
merge = foldr1 orElse
```


Conclusions

- `STM` provides a useful concurrent model which is more abstract that classics approaches.
- Reasoning about `SMT` programs seems easier than concurrent Haskell standards programs.
- The granularity in the concurrence is not quite clear.
- Deadlock is avoided (by construction).
- The behaviour of the model is so much dependent on the `SMT` implementation.
- The implementation of this technique in other paradigms is on research.

Implementation

- An low level layer is integrated to the Haskell runtime system, providing useful interface
- Using the low level layer a high level layer implements the datatype `SMT`.

An example of a bounded queue was used to measured the preformance of `SMT` against classic implementation with concurrent Haskell.

References

[1] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, Satnam Singh. *Lock Free Data Structures using STM in Haskell.*

[2] Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. *Composable Memory Transactions*. PPoP 2005. ACM.

[3] Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. *Concurrent Haskell*.