Universit degli Studi di Udine
Dipartimento di Scienze
Matematiche, Informatiche e Fisiche
Corso di Laurea in Informatica

Implementation of Open Transactional Memory in Haskell

Candidate:
Valentino Picotti

Supervisor:
Prof. Marino Miculan

Co-Supervisors:
Marco Peressotti
Nicola Gigante

Transactional Memory

Transactional Memory (TM) has emerged as a promising alternative to traditional *lock-based* synchronization.
Advantages of TM:

- Lock-free programming
- Avoids deadlocks, race conditions and priority inversions
- Composability and scalability
- Exploit multi-core architectures

Transactional Memory: the idea

The blocks of code are marked as *atomic* and their execution will appear either if it was performed instantaneously at some unique point in time, or, if aborted, as if it never happened.

The problem

Most TM models admit only *isolated* transactions, which are not adequate in multi-threaded programming where transactions have to interact via shared data *before* committing.

Example

A simple example is a request-response interaction via a shared buffer. Synchronization is required to regulate accesses to the buffer.

```
// Party1 (Master)
atomically {
  // put request in b
  up(c1);
  // some other code
  down(c2);
  // get answer from b
}
```

```
// Party2 (Worker)
atomically {
  // some code before
  down(c1);
  // get request from b
  // put answer in b
  up(c2);
  // some code after
}
```

Thesis proposal

The key observation is that *atomicity* and *isolation* should be seen as two disjoint computational aspects.

Thesis proposal

The key observation is that *atomicity* and *isolation* should be seen as two disjoint computational aspects.

- an atomic *non-isolated* block is executed “all-or-nothing”

Thesis proposal

The key observation is that *atomicity* and *isolation* should be seen as two disjoint computational aspects.

- an atomic *non-isolated* block is executed “all-or-nothing”
- an *isolated* block of code is executed “as it were the only one”

Open transactions

Atomic non-isolated blocks can be used for implementing safe composable interacting memory transactions, called *open transactions*.

	non-atomic	atomic
isolated	mutex block	TM
non-isolated	Normal block	Open transactions

OTM

Open Transactional Memory (OTM) is the model that implements *open transactions*.

OTM

Open Transactional Memory (OTM) is the model that implements *open transactions*. In this model:

- a transaction is composed by several threads, called *participants*

OTM

Open Transactional Memory (OTM) is the model that implements *open transactions*. In this model:

- a transaction is composed by several threads, called *participants*
- a transaction commits when all its participants commit, and aborts if any thread aborts

OTM

Open Transactional Memory (OTM) is the model that implements *open transactions*. In this model:

- a transaction is composed by several threads, called *participants*
- a transaction commits when all its participants commit, and aborts if any thread aborts
- accesses to shared data cause transactions to be *transparently merged* into a single one

OTM in Haskell

OTM is presented in the context of Haskell because its type system facilitate the reasoning on transactional memory:

- At the type system level we distinguish isolated atomic actions, represented as values of type *ITM a*, and non isolated atomic actions, as values of type *OTM a*.
- Actions can be sequentially composed preserving atomicity and, for ITM actions, isolation.

OTM interface: transactional memory

Transactional variables:

```
data OTVar a
```

Accesses to transactional variables:

```
-- Monad Transactional Memory
class (Monad m) => MonadTM m where
    newOTVar     :: a -> m (OTVar a)
    readOTVar    :: OTVar a -> m a
    writeOTVar   :: OTVar a -> a -> m ()
```

OTM interface: transactional memory

Example of an isolated update:

```
modifyOTVar :: OTVar a -> (a -> a) -> ITM ()
modifyOTVar var f = do
    x <- readOTVar var
    writeOTVar var (f x)
```

OTM interface: running transactions

Running isolated and atomic computations:

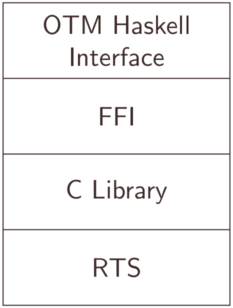
```
atomic   :: OTM a -> IO a
isolated :: ITM a -> OTM a
```

Equivalence with other TM implementations:

```
atomically = atomic . isolated
```

OTM implementation

Modules dependency:



OTM implementation: merge of transactions

- Each transaction records its participants in the transactional log.
- Merge of transactions are handled with an Union-find data structure.
- Union-by-rank and path compression heuristics are applied.

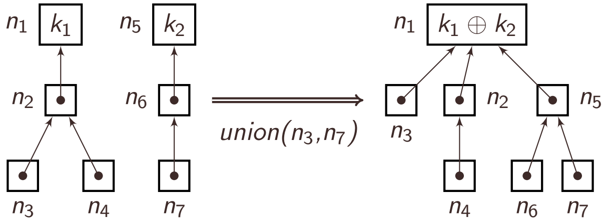


Figure: Merging two transactional log structures.

OTM implementation: validation of open transactions

The validation of open transactions is composed by two phases:

- *voting phase*: each *participant* votes for committing or aborting
- *agreement phase*: each *participant* commits or aborts depending on the outcome of the previous phase.

Conclusions

The OTM model separates isolated transactions from non-isolated ones.
The Haskell implementation is a conservative extension of STM:

- Semantically $\mathit{atomically} = \mathit{atomic}\ .\ \mathit{isolated}$
- From the user point of view, the libraries have the same interface
- From the implementation point of view, we provide the same interface of STM against the runtime system