Extending Software Transactional Memory in Clojure with Side-Effects and Transaction Control

Søren Kejser Jensen
Lone Leth Thomsen

Department of Computer Science
Aalborg University, Denmark
{skj,lone}@cs.aau.dk

Agenda

Introduction

Extensions

Implementation

Conclusion

Problem Definition

- The free lunch is over.
- Multi-core processors have become the default.
- Development of multi-threaded programs is a complicated task:
  - Deadlocks.
  - Race Conditions.
  - Non Determinism.

Contributions

- Extended a software transactional memory implementation, based on multi-version concurrency control, in a dynamic Lisp language with:
  - An interface for multiple methods for synchronising side-effects.
  - Known methods for transaction control with additional novel extensions.
- eClojure, an implementation of both extensions with unit tests in Clojure.
  - `https://github.com/skejserjensen/eclojure`

Clojure

- Clojure provides a good platform for developing multi-threaded programs:
  - Implemented on the Java Virtual Machine and Common Language Runtime.
  - Can interoperate with existing libraries developed for each platform.
  - Uses immutable data structures by default, facilitating concurrent access.
  - Provides software transactional memory to simplify sharing of mutable data.

Software Transactional Memory

- Alleviates the problem of locking order by restarting on conflicts.
- In Clojure implemented using multi-version concurrency control.
- Clojure's implementation of software-transactional memory provides:
  - Snapshot isolation.
  - Concurrent read and writes.
  - The possibility of write skew.

Software Transactional Memory Example

- Synchronised summation of `[1 2 3 4 5 6 7 8 9 10]` to the value 55.

```
(def shared-data (ref [1 2 3 4 5 6 7 8 9 10]))

(dosync
  (alter shared-data
         (fn [elements] (reduce + elements))))
```

Synchronisation of Side-Effects

- Software transactional memory cannot synchronise forms with side-effects.

```
(def keys-ref (ref []))
(def rows-ref (ref vector-of-rows))

(dosync
  (let [row (first (deref rows-ref))
        next-key (database-insert row)]
    (alter keys-ref conj next-key)
    (alter rows-ref rest)))
```

Controlling a transaction

- Clojure provides no capability for controlling a running transaction.

```
(try
  (dosync
    (throw TerminateException))
  (catch TerminateException te))
```

- Such functionality can be emulated but would add additional complexity.

Event Manager

```
(listen event-key event-fn & event-args)
(listen-with-params event-key thread-local
        delete-after-run event-fn & event-args)

(dismiss event-key event-fn dismiss-from)

(notify event-key)
(notify event-key context)

(context)
```

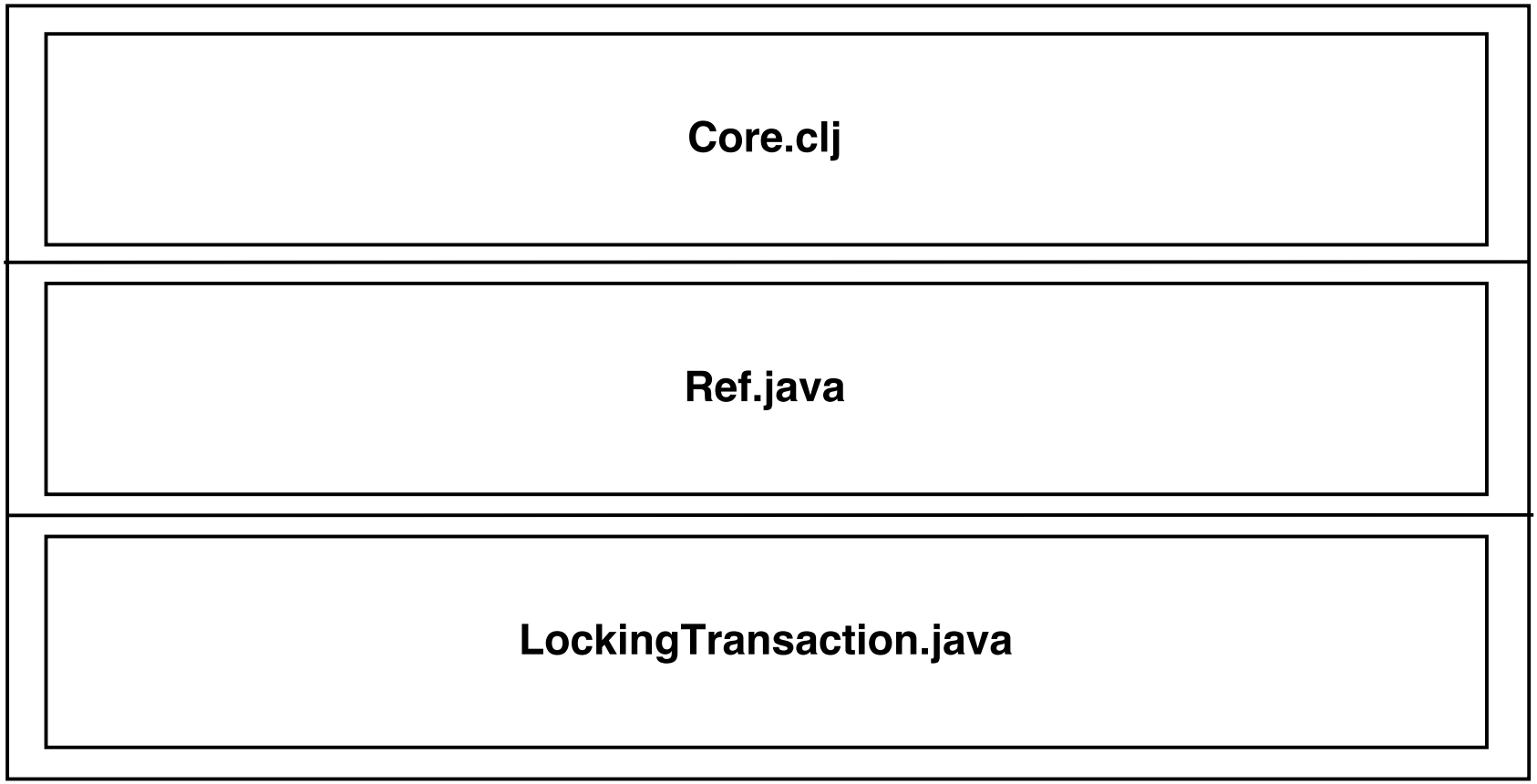
Synchronisation of Side-Effects

```
(after-commit & body)
(after-commit-fn event-fn & event-args)

(on-abort & body)
(on-abort-fn event-fn & event-args)

(on-commit & body)
(on-commit-fn event-fn & event-args)

(lock-refs func & body)

(java-ref x)
(java-ref x & options)

(alter-run input-ref func & args)
(commute-run input-ref func & args)
```

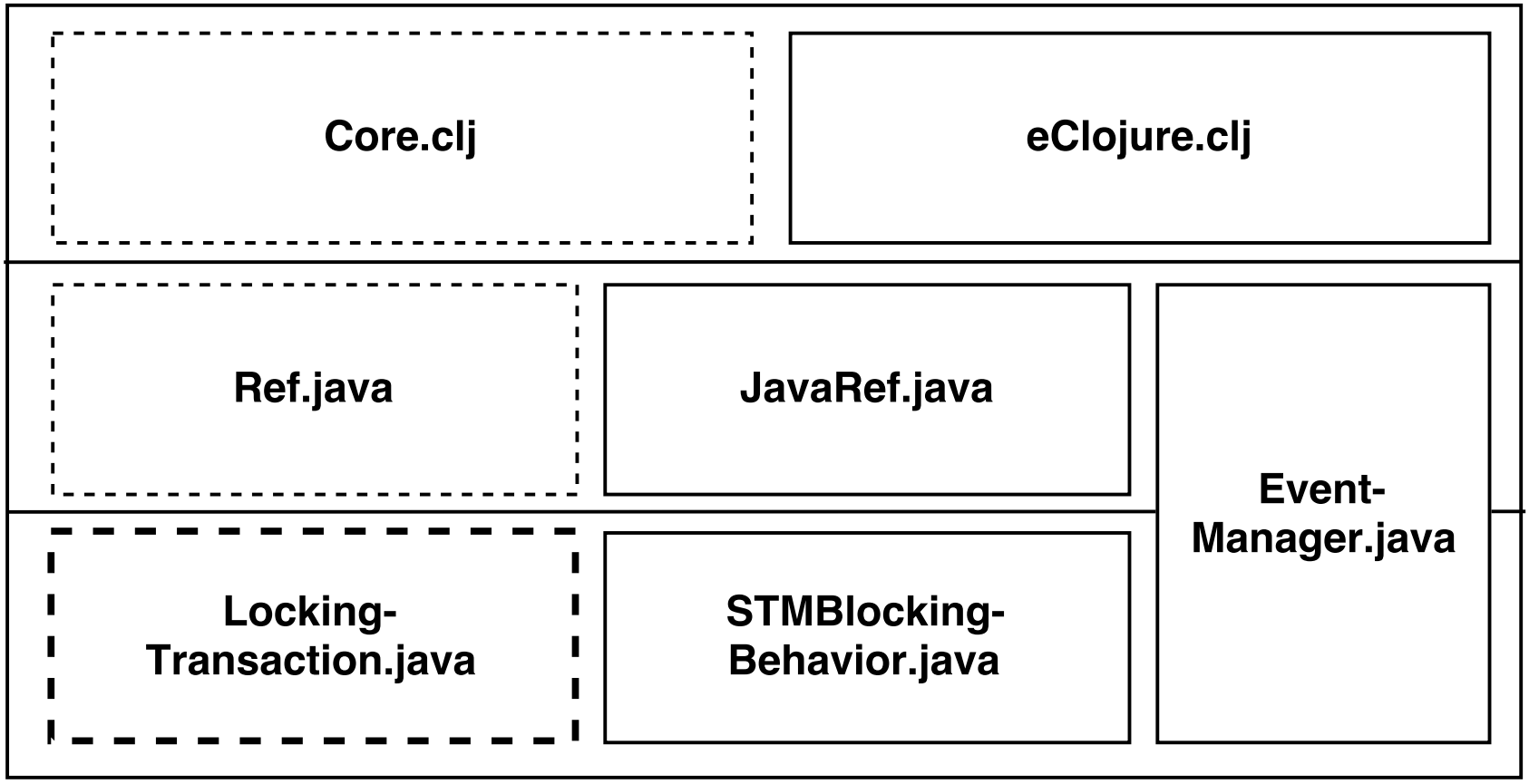
Transaction Control

```
(retry [])
(retry [refs])
(retry [refs func & args])

(retry-all [])
(retry-all [refs])
(retry-all [refs func & args])

(or-else & funcs)
(or-else-all & funcs)

(terminate)
```

Implementation of Software Transactional Memory in Clojure



Implementation of Software Transactional Memory in eClojure



Conclusion

- eClojure extends the implementation of software transactional memory in Clojure with support for synchronising side-effects and transaction control.
- Initial evaluation indicates a usability improvement at the cost of overhead.
- We anticipate the overhead could be further reduced through optimisation.
- Future work includes a larger evaluation, optimisation and automation.

Questions?