Abstract
This is the handout version talk about software transactional memory and automatic mutual exclusion. The wisdom is taken from [Abadi et al., 2008] and [Jagannathan et al., 2005], for the most part.

1 Introduction

Motivation
- concurrency ⇒ concurrency control
- nowaday’s languages: lock-based (good ol’ mutex)
- disadvantages:
  - low-level of abstraction
  - difficult to reason about
  - “conservative” protection ⇒ performance penalty / deadlocks
  - pessimistic approach to concurrency control
- here: “optimistic” approach
  - reduce crit-secs, more concurrency ⇒ non-blocking

Transactions
- coming from the data-base community
- control abstraction
- important correctness/failure properties: ACID transaction semantics = “illusion” of mutex
  1. atomicity
  2. isolation
  3. consistency
  4. durability
2 Transactional Java

TFJ

- taken from [Jagannathan et al., 2005]
- extending Featherweight Java with transactions
  - state
  - multi-threading (of course)
  - transactions
- featuring: nested and multi-threaded transactions
- operational semantics, 2 concretizations
  - versioning
  - 2-phase locking
- correctness proof: serializability

Why are transactions more high-level?

Listing 1: TFJ example

class Transactor {
  u: Updater;
  r: Runner;
  init (r: Runner, u: Updater) { this.u := u;
    this.r := r;
    this }
  run () {
    onacid
    this.u.update(); // write
    this.r.run(); // spawn intervening activity
    thus.u.val; // read
    commit
  }
}

Syntax

\[
P ::= 0 \mid P \parallel P \mid t(e) \quad \text{process}
L ::= \text{classC}\{f; M}\quad \text{class definition}
M ::= m(\vec{x})\{e\} \quad \text{method}
e ::= x \mid e.f \mid e.m(\vec{x}) \mid e.f := e \quad \text{expression}
| \quad \text{newsC} \mid \text{spawn e} \mid \text{onacid} \mid \text{commit} \mid \text{null}
v ::= r \mid v.f \mid v.m(\vec{v}) \mid b.f := v \quad \text{values/basic expressions}
\]

- basically 2 additions:
- **onacid**: start a transaction
- **commit**: end a transaction

**Semantics**

- given **operationally** (SOS, as usual . . .)
  - labelled transition system
  - evaluation-contexts
- 2 “stages”:
  1. first “general” semantics
  2. afterwards: 2 concretizations
- 2-level semantics
  1. **local** = per thread
  2. **global** = many threads

**2.1 Operational semantics without transactions**

**Underlying semantics: no transactions**

- for illustration here, only
- no separation in local ↔ global steps
- no transaction handling (but concurrency)
- **heap-manipulations** (read, write, extend) left “unspecified”
- configuration (local/global): \( \Gamma \vdash e \)

**Operational semantics: no transactions**
\[
\text{read}(r, \Gamma) = C(\vec{u}) \quad \text{fields}(C) = \vec{f} \\
\Gamma \vdash r.f_i : \Gamma' \\
\text{write}(r \mapsto C(\vec{r})) = \Gamma' \\
\text{mbody}(m, C) = (\vec{x}, e) \\
\Gamma \vdash r.m(\vec{r}) : \Gamma' \\
\text{new}(C()) = \Gamma' \\
\text{spawn}(n, E', \Gamma) = \Gamma' \\
\text{R-FIELD} \\
\text{R-ASSIGN} \\
\text{R-INVOK}E \\
\text{R-NEW} \\
\text{R-SPAWN}
\]

\[\begin{align*}
\text{read}(r, \Gamma) &= C(\vec{u}) \\
\text{write}(r \mapsto C(\vec{r})) &= \Gamma' \\
\text{mbody}(m, C) &= (\vec{x}, e) \\
\text{new}(C()) &= \Gamma' \\
\text{spawn}(n, E', \Gamma) &= \Gamma'
\end{align*}\]

2.2 Transactional semantics

Introducing transactions

- as said: syntax: \textcolor{red}{\textbf{onacid}} + \textcolor{blue}{\textbf{commit}}
- steps: split into 2 levels
  1. local: per thread
  2. global: “inter“-thread
- more complicated “memory model“
  - each thread has a local copy
  - how that exactly works ⇒ depending on the kind of transaction implementation (see later)
- general idea: optimistic approach
  - each thread works on its local copy (no locks, no regard of others)
  - local copy ⇒ isolation
  - when committing: check for conflicts ⇒
    * no: ⇒ make the effect visible
    * yes: ⇒ abort
Transactions and threads

- both are dynamic
  - thread creation by `spawn`
  - transaction “creation” by `onacid`

- transaction structure: nested
  - a transaction can contain inner transactions
  - child transactions must commit before outer transaction

- child transaction
  - ✴ commits ⇒ effects become visible to outer transaction
  - ✴ aborts ⇒ outer transaction does not abort

- relationship:
  - each thread inside an enclosing transaction
  - “multi” threads in one transaction

Local steps

- steps concerning one thread
- basic “single-threaded”, “non-transactional” steps
- local state/configuration:
  - “simple” expression $e + \text{local environment } E$

$E \vdash e$

- $E$
  - per transaction (labelled with $l$): local (partial) “state” = assoc of references to values
  - manipulated by read/write/extend
  - details determine the transactional model
  - Note: read-access may change $E$

---

1Thread structure: flat. One could make a hierarchical “father-child” structure, but it’s irrelevant here.
2or toplevel
3The paper itself is undecided whether to call it transaction environment or a sequence of transaction environments.
Local steps: rules

\[ \text{read}(r, E) = E', C(\vec{u}) \quad \text{fields}(C) = \vec{f} \]

\[ \text{R-FIELD} \]

\[ \mathcal{E} \vdash r.f_i \quad \mathcal{E}' \vdash u_i \]

\[ \text{write}(r \mapsto C(\vec{r}) \downarrow r', E') = E'' \]

\[ \text{R-ASSIGN} \]

\[ \mathcal{E} \vdash r.f_i = r' \quad \mathcal{E}' \vdash r' \]

\[ \text{mbody}(m, C) = (\bar{x}, e) \]

\[ \text{R-INVOK} \]

\[ \mathcal{E} \vdash r.m(\vec{r}) = e[\vec{x}/\vec{r}] [r/\text{this}] \]

\[ \text{R-NEW} \]

\[ \text{extend}(r \mapsto C(\text{null}), E) = \mathcal{E}' \]

\[ \text{R-NEW} \]

\[ r \text{ fresh} \]

\[ \mathcal{E} \vdash C(\vec{r}) \]

\[ \mathcal{E}' \vdash \vec{r} \]

\[ \mathcal{E} \vdash \text{new } C() \]

\[ \mathcal{E}' \vdash \vec{r} \]

Global steps

- behavior of \textbf{multiple interacting threads}

\[ n_1(e_1) \parallel \ldots \parallel n_k(e_k) = P \]

- \textbf{global state/configuration}

\[ \Gamma \vdash P \]

= program \( P \) + \textbf{global environment} \( \Gamma \) = local environment per thread:

\[ n_1: E_1, \ldots, n_k: E_k \vdash n_1(e_1) \ldots n_k(e_k) \]

- \textbf{transitions}

\[ \Gamma \vdash P \xrightarrow{\alpha} \Gamma' \vdash P' \]

Global steps: rules (1)

\[ P = P'' \parallel n(e) \quad \mathcal{E} \vdash e \quad \mathcal{E}' \vdash e' \quad P' = P'' \parallel n(e') \]

\[ \text{reflect}(n, \mathcal{E}', \Gamma) = \Gamma' \]

\[ \text{G-PLAIN} \]

\[ \Gamma \vdash P \xrightarrow{\alpha} \text{new } C() \]

\[ \Gamma' \vdash \text{new } C() \]

\[ \mathcal{E} \vdash \text{new } C() \]

\[ \mathcal{E}' \vdash \vec{r} \]

\[ \mathcal{E} \vdash \text{new } C() \]

\[ \mathcal{E}' \vdash \vec{r} \]

\[ \mathcal{E} \vdash \text{new } C() \]

\[ \mathcal{E}' \vdash \vec{r} \]

\[ \text{G-PLAIN} \]

\[ P = P'' \parallel n(E[\text{spawn } e]) \quad P' = P'' \parallel n(E[\text{null}]) \parallel n'(e') \]

\[ n' \text{ fresh} \quad \text{spawn}(n, E', \Gamma) = \Gamma' \]

\[ \text{G-SPAWN} \]

\[ \Gamma \vdash P \xrightarrow{\alpha} \text{new } C() \]

\[ \Gamma' \vdash \text{new } C() \]

\[ \mathcal{E} \vdash \text{new } C() \]

\[ \mathcal{E}' \vdash \vec{r} \]

\[ \mathcal{E} \vdash \text{new } C() \]

\[ \mathcal{E}' \vdash \vec{r} \]

\[ \mathcal{E} \vdash \text{new } C() \]

\[ \mathcal{E}' \vdash \vec{r} \]

\[ \text{G-SPAWN} \]

\[ P = P'' \parallel n(r) \quad \Gamma = n: E', \Gamma' \]

\[ \text{G-ThKill} \]

\[ \Gamma \vdash P \xrightarrow{\alpha} \text{new } C() \]

\[ \Gamma' \vdash \text{new } C() \]

\[ \text{G-ThKill} \]
Global steps: transaction handling

- **start** a transaction:
  - basically straightforward
  - create a new transaction label

- finish a transaction (**commit**):
  - “publish” the result
  - slightly more complex, because of *multi-threaded* transactions
  
  ⇒ *join* all threads that are about to commit the transaction in question
  
  - transaction in question: the “innermost” meant by the commit-action

Global steps: transaction rules (2)

\[
\begin{align*}
P = P'' \parallel n(E[onacid]) & \quad P' = P'' \parallel n(E[null]) \\
& \quad \text{\textit{l fresh}} \quad \text{start}(l, n, \Gamma) = \Gamma' \\
\Gamma \vdash \text{G-TRANS} & \quad n, \Gamma \vdash \text{start}(l, n, \Gamma) = \Gamma' \\

\begin{array}{c}
P = P'' \parallel n(E[commit]) \quad P' = P'' \parallel n(E[null]) \\
\Gamma = \Gamma'', n: E \quad \mathcal{E} = \mathcal{E}', l: \mathcal{E} \\
\text{commit}((\bar{n}, \mathcal{E}, \Gamma)) = \Gamma' \quad n_1: E_1, n_2: E_2, \ldots, n_k: E_k \in \Gamma \quad \mathcal{E}' = \mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_k \\
\end{array}
\end{align*}
\]

G-COMM

\[
\Gamma \vdash P \overset{\text{G-COMM}}{\Rightarrow} \Gamma'' \vdash P'
\]

2.3 Versioning semantics

Versioning semantics

- so far: the *core* has been left abstract
- one concretization of the general semantics
- concretization of the memory manipulations
- local environment \( E \)
  \[
l_1: \rho_1, \ldots, l_k: \rho_k
\]
- \( l \): transaction label
- \( \rho \):
  - log (of that transaction/of the given thread)
  - (part of the) dynamic context of the transaction \( l \)
- \( E \) is ordered
  - current enclosing one: on the right
  - reflects the nesting of transactions
Environment manipulations (local)

remember the local steps, for one thread \( E \vdash r \rightarrow E' \vdash r' \)

**read:** given a reference \( r \), find the assoc. value
- **look-up** the value for \( r \), not necessary in the innermost (= rightmost) transaction
- **log** the found value for the innermost transaction, i.e., copy/record it into that transactions log

**write:** similarly, the old value is logged locally, too

**extend:** similarly, no old value is logged (fresh reference)

Environment manipulation (local)

\[
\begin{align*}
E = E', l ; \varrho & \quad \text{findlast}(r, E) = C(\vec{r}) \quad E'' = E', l ; (\varrho, r \mapsto C(\vec{r})) \quad \text{E-READ} \\
E = E', l ; \varrho & \quad \text{read}(r, E) = E'', C(\vec{r}) \\
E = E', l ; \varrho & \quad \text{findlast}(r, E) = D(\vec{r}'') \quad E'' = E', l ; (\varrho, r \mapsto D(\vec{r}''), r \mapsto C(\vec{r})) \quad \text{E-WRITE} \\
E = E', l ; \varrho & \quad \text{write}(r \mapsto C(\vec{r}), E) = E'' \\
E = E', l ; \varrho & \quad \text{extend}(r \mapsto C(\vec{r}), E) = E'' \quad \text{E-EXTEND} \\
\Gamma = n : E, \Gamma' & \quad \Gamma'' = n' : E', \Gamma \quad \text{E-SPAWN}
\end{align*}
\]

Environment manipulation: for transactions

- 2 operations: **start** and **commit**

**start:**
- easy (“optimistic”)
- create a new label for the transaction
- start with an empty log for the new transaction

**commit:**
- more tricky.
- **propagate** (“reflect”) bindings from the transaction to the parent
- commit only, if no **conflict** is detected
- **conflict**: values used (r/w) in \( l \) must coincide with values as in parent transaction
Environment manipulation: transactions

\[ \Gamma = n \mathcal{E}, \Gamma' = (l(E, l)), \Gamma \]

\[ \text{start}(l, n, \Gamma) = \Gamma'' \] \hspace{1cm} E-START

\[ \text{commit}(\langle \rangle, \langle \rangle, \Gamma) = \Gamma \] \hspace{1cm} E-COMMIT1

\[ \mathcal{E} = \mathcal{E}', l, \varrho \]
\[ \text{readset}(\varrho, \langle \rangle) = \varrho' \quad \text{writeset}(\varrho, \langle \rangle) = \varrho'' \]
\[ \text{check}(\varrho', \mathcal{E}') \]
\[ \mathcal{E}' = \mathcal{E}'', l', \varrho''' \]
\[ \text{reflect}(n, (\mathcal{E}'', l', \varrho''', \varrho'''), \Gamma) = \Gamma' \]
\[ \text{commit}(\langle \rangle, \langle \rangle, \Gamma) = \Gamma'' \] \hspace{1cm} E-COMMIT2

Checking an environment

Modsets

\[ \text{readset}(\langle \rangle, \mathcal{J}) = \langle \rangle \]
\[ \varrho = u \mapsto C(\mathcal{J}) \quad u \notin \mathcal{R} \]
\[ \text{readset}(\varrho, \mathcal{R}) = u \mapsto C(\mathcal{J}), \varrho' \]
\[ \text{readset}(\varrho', \mathcal{R}) = \varrho' \]
\[ \text{writeset}(\langle \rangle, \mathcal{J}) = \langle \rangle \]
\[ \text{writeset}(\varrho, \varrho') = u \mapsto D(\mathcal{J}), \varrho'' \]
\[ \varrho?r \mapsto C(\mathcal{R}), \varrho'' \quad \text{writeset}(\varrho'', \varrho') = \varrho''' \quad r \mapsto C(\mathcal{R}) \neq \text{first}(r, \varrho') \]
\[ \text{writeset}(\varrho, \varrho') = u \mapsto D(\mathcal{J}), \varrho''' \]

2.4 Two-phase locking

Two-phase locking

- different instantiation of the general semantics, slight alteration
- based on locks
- pessimistic
- two phases:
1. first get hold of all the locks needed for a transaction
2. then release them again

- **strict**: all acquiring is done **before** all releasing.

**Two-phase locking transactional semantics**

- “slight” alteration of the previous one
- transaction & locks
  - objects **have** locks for protection
  - locks are **held** by **transactions**
  - enter a transaction: all locks held by transaction or **prefix** creating an object.
- to support locking
  - unique **transaction label** $l_L$
  - lock environment $\varrho_L$
- $\varrho$ stores **lock ownership** (per reference): which transactions hold the lock = sequence to reflected nesting
- given $l_1, l_2, \ldots, l_k$
- change of lock-ownership:
  - acquire by grabbing
  - commit by child, and propagate the lock upwards

**Environment manipulation with locks (local)**

\[
\begin{align*}
\mathcal{E} &= \mathcal{E}', l; \varrho & \\text{findlast}(r, \mathcal{E}) &= C(\bar{r}) \\
\mathcal{E}'' &= \mathcal{E}', l; (g, r \mapsto C(\bar{r})) & \\text{checklock}(r, \mathcal{E}) &= \top & \quad \text{E-READ} \\
\text{read}(r, \mathcal{E}) &= \mathcal{E}'', C(\bar{r}) \\
\mathcal{E} &= \mathcal{E}', l; \varrho & \\text{findlast}(r, \mathcal{E}) &= D(\bar{r}') \\
\mathcal{E}' &= \mathcal{E}'', l; \varrho & \mathcal{E}'' &= \mathcal{E}'', l; (g, r \mapsto D(\bar{r}')), r \mapsto C(\bar{r})) & \quad \text{E-WRITE} \\
\text{write}(r \mapsto C(\bar{r}), \mathcal{E}) &= \mathcal{E}'' \\
\mathcal{E} &= \mathcal{E}', l; \varrho & \text{acquirelock}(r, \mathcal{E}) = \mathcal{E}'', l; (g, r \mapsto C(\bar{r})) \\
\text{extend}(r \mapsto C(\bar{r}), \mathcal{E}) &= \mathcal{E}'' & \quad \text{E-EXTEND}
\end{align*}
\]

*Note the difference to multi-threaded Java*
Environment manipulation: transactions

\[
\begin{align*}
\Gamma = n;E, \Gamma' & \quad \Gamma'' = (l:(E, E();)), \Gamma \\
\text{start}(l, n, \Gamma) & = \Gamma'' & \text{E-START} \\
\text{commit}(\langle \rangle, \langle \rangle, \Gamma) & = \Gamma & \text{E-COMMIT}_1 \\
E & = l_L:q_L, E' & \hat{q}_L & = \text{release}(l(E), q_L) & E'' & = l_L:q_L', E' \\
\text{reflect}(n, (E'', l':q''', q'''), \Gamma) & = \Gamma' & \text{commit}(\bar{n}, \hat{E}, \Gamma') & = \Gamma'' & \text{E-COMMIT}_2
\end{align*}
\]

Further development in the paper

- After the formalization: prove some “soundness results”
  - ultimately: “ACID”, serialization
  - techniques: “permutation lemmas”

3 Conclusion

Further reading

- **wait-free** data structures
- old, related theoretical results: [Lipton, 1975]: theory of left/right movers
- [Herlihy and Wing, 1990]: linearizability for concurrent objects
- **futures** [Welc et al., 2005]
- transactions for Java [Garthwaite and Nettles, 1996]
- software transactional memory [Shavit and Toitu, 1995]
- **automatic mutual exclusion** [Abadi et al., 2008] and originally [Isard and Birell, 2007]
- and another POPL’08 paper?

References

References


