Software Transactional Memory &
Automatic Mutual Exclusion

Martin Steffen

University of Oslo, Norway

Oslo

Introduction

Transactional Java
  Operational semantics without transactions
  Transactional semantics
  Versioning semantics
  Two-phase locking

Automatic mutual exclusion

Conclusion
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
Introduction

Transactional Java
  Operational semantics without transactions
  Transactional semantics
  Versioning semantics
  Two-phase locking

Automatic mutual exclusion

Conclusion
• 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?

class Transactor {
    u: Updater;
    r: Runner;

    init (r: Runner, u: Updater) {
        this.u := u;
        this.r := r;
    }

    run () {
        this.u.update(); // write
        this.r.run(); // spawn intervening activity
        thus.u.n.val; // read
    }
}
Why are transactions more high-level?

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.n.val; // read
  
  commit
  }
}
Syntax

\[ P ::= 0 \mid P \parallel P \mid t\langle e \rangle \]  
\[ L ::= \text{class}C\{f, \vec{M}\} \]  
\[ M ::= m(\vec{x})\{e\} \]  
\[ e ::= x \mid e.f \mid e, m(\vec{e}) \mid e.f ::= e \]  
\[ \mid \text{news}C \mid \text{spawn} e \mid \text{onacid} \mid \text{commit} \mid \text{null} \]  
\[ v ::= r \mid v, f \mid v.m(\vec{v}) \mid b.f ::= v \]  

- 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
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): Γ ⊩ e
Operational semantics: no transactions

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

\[ \Gamma \vdash r.f_i \quad \text{read}(r, \Gamma) \quad \Gamma \vdash u_i \]

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

\[ \Gamma \vdash r.f_i := r' \quad \text{write}(r \mapsto C(\vec{r}) \downarrow' \; \Gamma) = \Gamma' \]

\[ \text{read}(r, \Gamma) = C(\vec{r}) \quad \text{mbody}(m, C) = (\vec{x}, e) \]

\[ \Gamma \vdash r.m(\vec{r}) \quad \text{read}(r, \Gamma) \quad \Gamma \vdash e[\vec{r}/\vec{x}][r/\text{this}] \]

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

\[ \Gamma \vdash \text{new } C() \quad \text{extend}(r \mapsto C(\text{null}), \Gamma) = \Gamma' \]

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

\[ n' \text{ fresh} \]
Introducing transactions

- as said: syntax: `onacid + 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

---

1 Thread structure: flat. One could make a hierarchical “father-child” structure, but it’s irrelevant here.
2 or toplevel
Local steps

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

- $\mathcal{E}$:
  - 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 $\mathcal{E}$

---

$\text{3The paper itself is undecided whether to call it transaction environment or a sequence of transaction environments.}$
Local steps: rules

\[
\begin{align*}
\text{read}(r, \mathcal{E}) &= \mathcal{E}', C(\bar{u}) \quad \text{fields}(C) = \bar{f} & \text{R-FIELD} \\
\mathcal{E} &\vdash r.f_i \quad \text{rd } r \quad \mathcal{E}' \vdash u_i \\
\text{read}(r, \mathcal{E}) &= \mathcal{E}', C(\bar{r}) \quad \text{write}(r \mapsto C(\bar{r}) \downarrow_{i}', \mathcal{E}') = \mathcal{E}'' & \text{R-ASSIGN} \\
\mathcal{E} &\vdash r.f_i := r' \quad \text{wr } rr' \quad \mathcal{E}'' \vdash r' \\
\text{read}(r, \mathcal{E}) &= \mathcal{E}', C(\bar{r}) \quad \text{mbody}(m, C) = (\bar{x}, e) & \text{R-INVOKE} \\
\mathcal{E} &\vdash r.m(\bar{r}) \quad \text{rd } r \quad \mathcal{E}' \vdash e[\bar{r}/\bar{x}][r/\text{this}] \\
r \text{ fresh} \quad \text{extend}(r \mapsto C(\text{null}), \mathcal{E}) = \mathcal{E}' & \text{R-NEW} \\
\mathcal{E} &\vdash \text{new } C() \quad \text{xt } r \quad \mathcal{E}' \vdash r
\end{align*}
\]
Global steps

- behavior of multiple interacting threads

\[ n_1\langle e_1 \rangle \parallel \ldots \parallel n_k\langle e_k \rangle = P \]

- global state/configuration

\[ \Gamma \vdash P \]

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

\[ n_1: \mathcal{E}_1, \ldots n_k: \mathcal{E}_k \vdash n_1\langle e_1 \rangle \ldots n_k\langle e_k \rangle \]

- transitions

\[ \Gamma \vdash P \xrightarrow{\alpha} n\Gamma' \vdash P' \]
Global steps: rules (1)

\[
P = P'' \parallel n\langle e \rangle \quad \mathcal{E} \vdash e \xrightarrow{\alpha} \mathcal{E}' \vdash e' \quad P' = P'' \parallel n\langle e' \rangle
\]

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

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

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

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

\[
\Gamma \vdash P \xrightarrow{\text{sp } n'} n\Gamma' \vdash P'
\]

\[
P = P' \parallel n\langle r \rangle \quad \Gamma = n: \mathcal{E}, \Gamma'
\]

\[
\Gamma \vdash P \xrightarrow{k_i} n\Gamma' \vdash P''
\]
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)

\[ P = P'' \parallel n\langle E[\text{onacid}]\rangle \quad P' = P'' \parallel n\langle E[\text{null}]\rangle \]

\[ \text{fresh} \quad \text{start}(l, n, \Gamma) = \Gamma' \]

\[ \Gamma \vdash P \xrightarrow{ac} n\Gamma' \vdash P' \quad \text{G-TRANS} \]

\[ P = P'' \parallel n\langle E[\text{commit}]\rangle \quad P' = P'' \parallel n\langle E[\text{null}]\rangle \]

\[ \Gamma = \Gamma'', \text{ } n:\mathcal{E} \quad \mathcal{E} = \mathcal{E}', \text{ } l:\emptyset \quad \text{intranse}(l, \Gamma) = \tilde{n} = n_1 \ldots n_k \]

\[ \text{commit}(\tilde{n}, \tilde{\mathcal{E}}, \Gamma) = \Gamma' \quad n_1:\mathcal{E}_1, n_2:\mathcal{E}_2, \ldots n_k:\mathcal{E}_k \in \Gamma \quad \tilde{\mathcal{E}} = \mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_k \]

\[ \Gamma \vdash P \xrightarrow{co} n\Gamma' \vdash P' \quad \text{G-COM} \]
Versioning semantics

- so far: the **core** has been **left abstract**
- one **concretization** of the general semantics
- concretization of the **memory manipulations**
- local environment $\mathcal{E}$

$$l_1:\varrho_1, \ldots l_k:\varrho_k$$

- $l$: transaction label
- $\varrho$:
  - log (of that transaction/of the given thread)
  - (part of the) dynamic context of the transaction $l$
- $\mathcal{E}$ is **ordered**, 
  - current enclosing one: on the **right**
  - reflects the **nesting** of transactions
Environment manipulations (local)

remember the local steps, for one thread
\[ \mathcal{E} \vdash r \rightarrow \mathcal{E}' \vdash r' \]

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)

\[
\begin{align*}
\mathcal{E} & = \mathcal{E}', \ l : \varnothing \\
\overset{\text{findlast}(r, \mathcal{E}) = C(\vec{r})}{\text{read}(r, \mathcal{E}) = \mathcal{E}'', C(\vec{r})} & \quad \text{E-READ} \\
\mathcal{E} & = \mathcal{E}', \ l : \varnothing \\
\overset{\text{findlast}(r, \mathcal{E}) = D(\vec{r}')}{\text{write}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}''} & \quad \text{E-WRITE} \\
\mathcal{E} & = \mathcal{E}', \ l : \varnothing \\
\overset{\mathcal{E}'' = \mathcal{E}', l : (\varnothing, r \mapsto C(\vec{r}))}{\text{extend}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}''} & \quad \text{E-EXTEND} \\
\Gamma & = n : \mathcal{E}, \Gamma' \\
\overset{\Gamma'' = n' : \mathcal{E}', \Gamma}{\text{spawn}(n, n', \Gamma) = \Gamma''} & \quad \text{E-SPAWN}
\end{align*}
\]
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 / must coincide with values as in parent transaction
Environment manipulation: transactions

\[ \Gamma = n : \mathcal{E}, \Gamma' \quad \Rightarrow \quad \Gamma'' = (l : (\mathcal{E}, l : \langle \rangle)), \Gamma \quad \text{E-START} \]

\[ \text{start}(l, n, \Gamma) = \Gamma'' \]

\[ \text{E-COMMIT}_1 \]

\[ \text{commit}(\langle \rangle, \langle \rangle, \Gamma) = \Gamma \]

\[ \mathcal{E} = \mathcal{E}', l : \varrho \quad \text{readset}(\varrho, \langle \rangle) = \varrho' \quad \text{writeset}(\varrho, \langle \rangle) = \varrho'' \]

\[ \text{check}(\varrho', \mathcal{E}') \quad \mathcal{E}' = \mathcal{E}'', l' : \varrho''' \quad \text{reflect}(n, (\mathcal{E}'', l' : \varrho''', \varrho''), \Gamma) = \Gamma' \]

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

\[ \text{E-COMMIT}_2 \]

\[ \text{commit}(n \vec{n}, \mathcal{E} \vec{\mathcal{E}}, \Gamma) = \Gamma'' \]
Checking an environment
Modsets
Modsets

\[ \text{readset}(\langle \rangle, -) = \langle \rangle \]

\[ \varrho = u \mapsto C(\bar{u}) \quad u \not\in \bar{r} \quad \text{readset}(\varrho'', \bar{r}u) = \varrho' \]

\[ \text{readset}(\varrho, \bar{r}) = u \mapsto C(\bar{u}), \varrho' \]

\[ \varrho = u \mapsto C(\bar{u}), \varrho'' \quad u \in \bar{r} \quad \text{readset}(\varrho'', \bar{r}) = \varrho' \]

\[ \text{readset}(\varrho, \bar{r}) = \varrho' \]

\[ \text{writeset}(\langle \rangle, -) = \langle \rangle \]

\[ \varrho?r \mapsto C(\bar{r}), \varrho'' \quad \text{writeset}(\varrho'', \varrho') = \varrho''' \quad r \mapsto C(\bar{r}) \neq \text{first}(r, \varrho') \]

\[ \text{writeset}(\varrho, \varrho') = u \mapsto D(\bar{u}), \varrho''' \]
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\(^4\).
  - 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

\(^4\)Note the difference to multi-threaded Java
Environment manipulation with locks (local)

\[ \mathcal{E} = \mathcal{E}', \; l: \emptyset \quad \text{findlast}(r, \mathcal{E}) = C(\vec{r}) \]
\[ \mathcal{E}'' = \mathcal{E}', \; l: (\emptyset, r \mapsto C(\vec{r})) \quad \text{checklock}(r, \mathcal{E}) = \top \]
\[ \text{read}(r, \mathcal{E}) = \mathcal{E}''', \; C(\vec{r}) \]

E-READ

\[ \text{findlast}(r, \mathcal{E}) = D(\vec{r}') \quad \mathcal{E}' = \text{acquirelock}(r, \mathcal{E}) \]
\[ \mathcal{E}' = \mathcal{E}'', \; l: \emptyset \quad \mathcal{E}''' = \mathcal{E}'', \; l: (\emptyset, r \mapsto D(\vec{r}'), \; r \mapsto C(\vec{r})) \]
\[ \text{write}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}''' \]

E-WRITE

\[ \text{acquirelock}(r, \mathcal{E}) = \mathcal{E}', \; l: \emptyset \quad \mathcal{E}'' = \mathcal{E}', \; l: (\emptyset, r \mapsto C(\vec{r})) \]
\[ \text{extend}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}'' \]

E-EXTEND
Environment manipulation: transactions

\[
\Gamma = n : \mathcal{E}, \Gamma' \quad \Gamma'' = (l : (\mathcal{E}, l : \langle \rangle)), \Gamma \\
\text{E-START} \\
\text{start}(l, n, \Gamma) = \Gamma''
\]

\[
\text{E-COMMIT}_1 \\
\text{commit}(\langle \rangle, \langle \rangle, \Gamma) = \Gamma
\]

\[
\mathcal{E} = l_L : \varrho_L, \mathcal{E}' \\
\varrho_L' = \text{release}(l(\mathcal{E}), \varrho_L) \\
\mathcal{E}'' = l_L : \varrho_L', \mathcal{E}' \\
\text{reflect}(n, (\mathcal{E}'', l' : \varrho''', \varrho''), \Gamma) = \Gamma' \\
\text{commit}(\vec{n}, \vec{\mathcal{E}}, \Gamma') = \Gamma'' \\
\text{E-COMMIT}_2 \\
\text{commit}(n \vec{n}, \mathcal{E} \vec{\mathcal{E}}, \Gamma) = \Gamma''
\]
Further development in the paper

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

Transactional Java
  Operational semantics without transactions
  Transactional semantics
  Versioning semantics
  Two-phase locking

Automatic mutual exclusion

Conclusion
Automatic mutex

- See [Abadi et al., 2008]
- building on the “AME” proposal of [Isard and Birell, 2007]
- weak vs. strong atomicity:

Weak vs. strong

How does non-transactional code interacts with transactional?

- cf. Java’s synchronized-method
- important for library code, “instrumentation”
- user expectation, subtle errors
- weak atomicity more common/easier
AME calculus

- simple core-calc.
  - higher-order functions
  - heap/imperative features
  - concurrency via async
- protection by default
- “fragmentation” by user-command unprotected/“yield”
- cf. suspend-command in Creol

\(^5\)of course
AME syntax

\[
\begin{align*}
\nu & ::= \ c \mid x \mid \lambda x.e \\
\nc & ::= \ \text{unit} \mid \text{false} \mid \text{true} \\
\se & ::= \ \nu \quad \text{expressions: values} \\
& \quad \mid \ e \ e \quad \text{application} \\
& \quad \mid \ \text{ref} \ e \mid \ !e \mid \ e := e \\
& \quad \mid \ \text{async} \ e \\
& \quad \mid \ \text{blockuntil} \ e \\
& \quad \mid \ \text{unprotected} \ e
\end{align*}
\]
Strong semantics

- reference semantics
- evaluation style definition (eval. contexts slightly complicated)
- separation of protected and unprotected code
- configuration
  \[ \langle \sigma, T, e \rangle \]

1. heap \( \sigma \)
2. pool of expr’s/threads \( T \)
3. active expression \( e \)
Evaluation contexts

\[ \mathcal{P} ::= \emptyset | \mathcal{P} \ e | \text{ref} \mathcal{P} | \mathcal{P} := e | r := \mathcal{P} | \text{blockuntil} \ \mathcal{P} \]

\[ \mathcal{U} ::= \text{unprotected} \mathcal{E} | \mathcal{U} \ e | \nu \mathcal{U} | \text{ref} \mathcal{U} | \mathcal{U} | \mathcal{U} := e | r := \mathcal{U} | \text{blockuntil} \ \mathcal{U} \]

\[ \mathcal{E} ::= \emptyset | \mathcal{E} \ e | \nu \mathcal{E} | \text{ref} \mathcal{E} | \mathcal{E} | \mathcal{E} := e | r := \mathcal{E} | \text{blockuntil} \ \mathcal{E} | \text{unprotected} \mathcal{E} \]

\[ \mathcal{F} ::= \mathcal{T}.\mathcal{U}.\mathcal{T}', \text{unit} | \mathcal{T}, \mathcal{P} \]
\[ \langle \sigma, \mathcal{F}[(\lambda x.e) \ v]\rangle \rightarrow \langle \sigma, \mathcal{F}[e[v/x]]\rangle \quad \text{T-APP} \]

\[
\begin{array}{c}
\text{r fresh} \\
\hline
\langle \sigma, \mathcal{F}\text{ref } v\rangle \rightarrow \langle \sigma[r \mapsto v], \mathcal{F}r\rangle
\end{array} \quad \text{T-REF}
\]

\[
\begin{array}{c}
\sigma(r) = v \\
\hline
\langle \sigma, \mathcal{F}\text{!r}\rangle \rightarrow \langle \sigma, \mathcal{F}v\rangle
\end{array} \quad \text{T-DEREF}
\]

\[
\begin{array}{c}
\langle \sigma, \mathcal{F}r := v\rangle \rightarrow \langle \sigma[r \mapsto v], \mathcal{F}\text{unit}\rangle
\end{array} \quad \text{T-SET}
\]

\[
\begin{array}{c}
\langle \sigma, \mathcal{F}\text{async } e\rangle \rightarrow \langle \sigma, e.\mathcal{F}\text{unit}\rangle
\end{array} \quad \text{T-ASYNC}
\]

\[
\langle \sigma, \mathcal{F}\text{blockuntil true}\rangle \rightarrow \langle \sigma, \mathcal{F}\text{unit}\rangle \quad \text{T-BOCK}
\]

\[
\langle \sigma, T.\mathcal{P}[\text{unprotected } e], \text{unit}\rangle \rightarrow \langle \sigma, T.\mathcal{P}[\text{unprotected } e], \text{unit}\rangle \quad \text{T-UNPROTECT}
\]

\[
\langle \sigma, T.\mathcal{E}[\text{unprotected } v].T', \text{unit}\rangle \rightarrow \langle \sigma, T.\mathcal{E}[v].T', \text{unit}\rangle \quad \text{T-CLOSE}
\]

\[
\langle \sigma, T.e.T', \text{unit}\rangle \rightarrow \langle \sigma, T.T', e\rangle \quad \text{T-ACTIVATE}
\]
example: yielding

yield $\triangleq$ unprotected unit
Weak semantics

• more complex
• two variants
  • with roll-back
  • “optimistic”
• $\langle \sigma, T, e, f, l, P \rangle$
• interplay of transacted/non-transacted code can be tricky
Examples

\[
\begin{align*}
  r_1 & := x ; \\
  r_2 & := x ;
\end{align*}
\]

unprotected \{ 
  \begin{align*}
    x & := 1 \\
  \end{align*}
\}
r1 := u
r2 := v
if (r1 != r2) {
    x := 42;
}

is there a race?
is there a **race**?

- intuitively: no race
Results

- weak = strong semantics, under certain restrictions
- violation-freedom, separation
- generalization of race-freedom
- type and effect system for separation

\(^6\) race freedom is not enough
Introduction

Transactional Java
Operational semantics without transactions
Transactional semantics
Versioning semantics
Two-phase locking

Automatic mutual exclusion

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?
- [Blundell et al., 2006]
- language extensions with transactions (often based on Java): [Carlstrom et al., 2006] [Harris and Fraser, 2003], Haskell, Caml, Lisp, Fortress, X10, ...
References

Semantics of transactional memory and automatic mutual exclusion.
In *Proceedings of POPL ’08*. ACM.

Subtelties of transactional memory atomicity semantics.

The ATOMOS transactional programming language.
In *ACM Conference on Programming Language Design and Implementation (Ottawa, Ontario, Canada)*. ACM.

Transactions for Java.

The transactional memory / garbage collection analogy.
In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) ’97*. ACM.
In *SIGPLAN Notices*.

[Harris and Fraser, 2003] Harris, T. and Fraser, K. (2003).
Language support for lightweight transactions.
In *Eighteenth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) ’03*. ACM.
In *SIGPLAN Notices*.

Linearizability: A Correctness Condition for Concurrent Objects.
*ACM Transactions on Programming Languages and Systems*, 12(3):463–492.

Automatic mutual exclusion.
In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*. 
A transactional object calculus.

Reduction: A method of proving properties of parallel programs.
Papers from the Second ACM Symposium on POPL, Palo Alto, California.

Software transactional memory.

Safe futures in Java.
In *Twentieth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) ’05*, pages 439 – 453. ACM.
In *SIGPLAN Notices*.