Software Transactional Memory & Automatic Mutual Exclusion

Martin Steffen

University of Oslo, Norway

Oslo

10. Feb. + 3. March 2009

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

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?

```
class Transactor {
 u: Updater;
 r: Runner;
 init (r: Runner , u: Updater ) { this.u := u;
                                  this.r := r;
                                  this }
 run () {

     this.u.update();              // write
     this.r.run();                 // spawn interve
     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 interve
     thus.u.n.val;                // read
   commit
 }
}
```

Syntax

$$
\begin{array}{rcll}
P & ::= & 0 \mid P \parallel P \mid t\langle e \rangle & \text{process} \\
L & ::= & class C\{\vec{f}; \vec{M}\} & \text{class definition} \\
M & ::= & m(\vec{x})\{e\} & \text{method} \\
e & ::= & x \mid e.f \mid e, m(\vec{e}) \mid e.f := e & \text{expression} \\
  & \mid & news C \mid spawn\ e \mid onacid \mid commit \mid null & \\
v & ::= & r \mid v, f \mid v.m(\vec{v}) \mid b.f := v & \text{values/basic exp}
\end{array}
$$

- 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 $\leftrightarrow$ global steps
- no transaction handling (but concurrency)
- heap-manipulations (read, write, extend) left "unspecified"
- configuration (local/global): $\Gamma \vdash e$

Operational semantics: no transactions

$$\frac{read(r,\Gamma) = C(\vec{u}) \qquad fields(C) = \vec{f}}{\Gamma \vdash r.f_i \overset{rd\ r}{\longrightarrow} \Gamma \vdash u_i} \text{ R-FIELD}$$

$$\frac{read(r,\Gamma) = C(\vec{r}) \qquad write(r \mapsto C(\vec{r}) \downarrow_i^{r'}, \Gamma) = \Gamma'}{\Gamma \vdash r.f_i := r' \overset{wr\ rr'}{\longrightarrow} \Gamma' \vdash r'} \text{ R-ASSIGN}$$

$$\frac{read(r,\Gamma) = C(\vec{r}) \qquad mbody(m, C) = (\vec{x}, e)}{\Gamma \vdash r.m(\vec{r}) \overset{rd\ r}{\longrightarrow} \Gamma \vdash e[\vec{r}/\vec{x}][r/this]} \text{ R-INVOKE}$$

$$\frac{r\ fresh \qquad extend(r \mapsto C(\overrightarrow{\text{null}}), \Gamma) = \Gamma'}{\Gamma \vdash \text{new } C() \overset{xt\ r}{\longrightarrow} \Gamma' \vdash r} \text{ R-NEW}$$

$$P = P'' \parallel n\langle E[\text{spawn } e]\rangle \qquad P' = P'' \parallel n\langle E[\text{null}]\rangle \parallel n'\langle e'\rangle$$
$$n'\ fresh$$
R-SPAWN

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[1]
  - 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[2]
  - " 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 + local environment $\mathcal{E}$ [3]

$$\mathcal{E} \vdash e$$

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

[3] The paper itself is undecided whether to call it transaction environment or a sequence of transaction environments.

Local steps: rules

$$\frac{read(r, \mathcal{E}) = \mathcal{E}', C(\vec{u}) \qquad fields(C) = \vec{f}}{\mathcal{E} \vdash r.f_i \xrightarrow{rd\ r} \mathcal{E}' \vdash u_i} \text{R-FIELD}$$

$$\frac{read(r, \mathcal{E}) = \mathcal{E}', C(\vec{r}) \qquad write(r \mapsto C(\vec{r}) \downarrow_i^{r'}, \mathcal{E}') = \mathcal{E}''}{\mathcal{E} \vdash r.f_i := r' \xrightarrow{wr\ rr'} \mathcal{E}'' \vdash r'} \text{R-ASSIGN}$$

$$\frac{read(r, \mathcal{E}) = \mathcal{E}', C(\vec{r}) \qquad mbody(m, C) = (\vec{x}, e)}{\mathcal{E} \vdash r.m(\vec{r}) \xrightarrow{rd\ r} \mathcal{E}' \vdash e[\vec{r}/\vec{x}][r/this]} \text{R-INVOKE}$$

$$\frac{r\ fresh \qquad extend(r \mapsto C(\vec{\text{null}}), \mathcal{E}) = \mathcal{E}'}{\mathcal{E} \vdash \text{new}\ C() \xrightarrow{xt\ r} \mathcal{E}' \vdash r} \text{R-NEW}$$

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 Γ = 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

$$\Gamma \vdash P \stackrel{\alpha}{\Longrightarrow}_n \Gamma' \vdash P'$$

Global steps: rules (1)

$$\frac{P = P'' \parallel n\langle e\rangle \qquad \mathcal{E} \vdash e \xrightarrow{\alpha} \mathcal{E}' \vdash e' \qquad P' = P'' \parallel n\langle e'\rangle \qquad reflect(n, \mathcal{E}', \Gamma) = \Gamma'}{\Gamma \vdash P \overset{\alpha}{\Longrightarrow}_n \Gamma' \vdash P'} \text{ G-PLAIN}$$

$$\frac{P = P'' \parallel n\langle E[\mathrm{spawn}\ e]\rangle \qquad P' = P'' \parallel n\langle E[\mathrm{null}]\rangle \parallel n'\langle e'\rangle \qquad n' \textit{ fresh} \qquad spawn(n, \mathcal{E}', \Gamma) = \Gamma'}{\Gamma \vdash P \overset{sp\ n'}{\Longrightarrow}_n \Gamma' \vdash P'} \text{ G-SPAWN}$$

$$\frac{P = P' \parallel n\langle r\rangle \qquad \Gamma = n{:}\mathcal{E}, \Gamma'}{\Gamma \vdash P \overset{ki}{\Longrightarrow}_n \Gamma' \vdash P''} \text{ G-THKILL}$$

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)

$$\frac{P = P'' \parallel n\langle E[\text{onacid}]\rangle \qquad P' = P'' \parallel n\langle E[\text{null}]\rangle \qquad l \textit{ fresh} \qquad \textit{start}(l, n, \Gamma) = \Gamma'}{\Gamma \vdash P \overset{ac}{\Longrightarrow}_n \Gamma' \vdash P'} \text{ G-TRANS}$$

$$\frac{\begin{array}{c} P = P'' \parallel n\langle E[\overrightarrow{\text{commit}}]\rangle \qquad P' = P'' \parallel n\langle E[\vec{\text{null}}]\rangle \\ \Gamma = \Gamma'', n{:}\mathcal{E} \qquad \mathcal{E} = \mathcal{E}', l{:}\varrho \qquad \textit{intranse}(l, \Gamma) = \vec{n} = n_1 \ldots n_k \\ \textit{commit}(\vec{n}, \vec{\mathcal{E}}, \Gamma) = \Gamma' \quad n_1{:}\mathcal{E}_1, n_2{:}\mathcal{E}_2, \ldots n_k{:}\mathcal{E}_k \in \Gamma \quad \vec{\mathcal{E}} = \mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_k \end{array}}{\Gamma \vdash P \overset{co}{\Longrightarrow}_n \Gamma' \vdash P'} \text{ G-CO}$$

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
- ϱ:
  - 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

$\mathcal{E} \vdash r \rightharpoonup \mathcal{E}' \vdash r'$

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)

$$\frac{\mathcal{E} = \mathcal{E}', l{:}\varrho \qquad \mathit{findlast}(r, \mathcal{E}) = C(\vec{r}) \qquad \mathcal{E}'' = \mathcal{E}', l{:}(\varrho, r \mapsto C(\vec{r}))}{\mathit{read}(r, \mathcal{E}) = \mathcal{E}'', C(\vec{r})} \text{ E-READ}$$

$$\frac{\mathcal{E} = \mathcal{E}', l{:}\varrho \quad \mathit{findlast}(r, \mathcal{E}) = D(\vec{r}') \quad \mathcal{E}'' = \mathcal{E}', l{:}(\varrho, r \mapsto D(\vec{r}'), r \mapsto C(\vec{r}))}{\mathit{write}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}''} \text{ E-V}$$

$$\frac{\mathcal{E} = \mathcal{E}', l{:}\varrho \qquad \mathcal{E}'' = \mathcal{E}', l{:}(\varrho, r \mapsto C(\vec{r}))}{\mathit{extend}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}''} \text{ E-EXTEND}$$

$$\frac{\Gamma = n{:}\mathcal{E}, \Gamma' \qquad \Gamma'' = n'{:}\mathcal{E}', \Gamma}{\mathit{spawn}(n, n', \Gamma) = \Gamma''} \text{ E-SPAWN}$$

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

$$\frac{\Gamma = n{:}\mathcal{E}, \Gamma' \qquad \Gamma'' = (l{:}(\mathcal{E}, l{:}\langle\rangle)), \Gamma}{\mathit{start}(l, n, \Gamma) = \Gamma''} \text{ E-START}$$

$$\frac{}{\mathit{commit}(\langle\rangle, \langle\rangle, \Gamma) = \Gamma} \text{ E-COMMIT}_1$$

$$\frac{\begin{array}{c}\mathcal{E} = \mathcal{E}', l{:}\varrho \qquad \mathit{readset}(\varrho, \langle\rangle) = \varrho' \qquad \mathit{writeset}(\varrho, \langle\rangle) = \varrho'' \\ \mathit{check}(\varrho', \mathcal{E}') \qquad \mathcal{E}' = \mathcal{E}'', l'{:}\varrho''' \qquad \mathit{reflect}(n, (\mathcal{E}'', l'{:}\varrho''', \varrho''), \Gamma) = \Gamma' \\ \mathit{commit}(\vec{n}, \vec{\mathcal{E}}, \Gamma') = \Gamma''\end{array}}{\mathit{commit}(n\ \vec{n}, \mathcal{E}\ \vec{\mathcal{E}}, \Gamma) = \Gamma''} \text{ E-CO}$$

Checking an environment

Modsets

Modsets

$$\frac{}{\mathit{readset}(\langle\rangle, \_) = \langle\rangle}$$

$$\frac{\varrho = u \mapsto C(\vec{u}) \qquad u \notin \vec{r} \qquad \mathit{readset}(\varrho'', \vec{r}u) = \varrho'}{\mathit{readset}(\varrho, \vec{r}) = u \mapsto C(\vec{u}), \varrho'}$$

$$\frac{\varrho = u \mapsto C(\vec{u}), \varrho'' \qquad u \in \vec{r} \qquad \mathit{readset}(\varrho'', \vec{r}) = \varrho'}{\mathit{readset}(\varrho, \vec{r}) = \varrho'}$$

$$\frac{}{\mathit{writeset}(\langle\rangle, \_) = \langle\rangle}$$

$$\frac{\varrho ? r \mapsto C(\vec{r}), \varrho'' \qquad \mathit{writeset}(\varrho'', \varrho') = \varrho''' \qquad r \mapsto C(\vec{r}) \neq \mathit{first}(r, \varrho')}{\mathit{writeset}(\varrho, \varrho') = u \mapsto D(\vec{u}), \varrho'''}$$

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 ϱ_L.
- ϱ 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)

$$\frac{\mathcal{E} = \mathcal{E}', l{:}\varrho \qquad \mathit{findlast}(r, \mathcal{E}) = C(\vec{r}) \qquad \mathcal{E}'' = \mathcal{E}', l{:}(\varrho, r \mapsto C(\vec{r})) \qquad \mathit{checklock}(r, \mathcal{E}) = \top}{\mathit{read}(r, \mathcal{E}) = \mathcal{E}'', C(\vec{r})} \text{ E-READ}$$

$$\frac{\mathit{findlast}(r, \mathcal{E}) = D(\vec{r}') \qquad \mathcal{E}' = \mathit{acquirelock}(r, E) \qquad \mathcal{E}' = \mathcal{E}'', l{:}\varrho \qquad \mathcal{E}''' = \mathcal{E}'', l{:}(\varrho, r \mapsto D(\vec{r}'), r \mapsto C(\vec{r}))}{\mathit{write}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}'''} \text{ E-WRITE}$$

$$\frac{\mathit{acquirelock}(r, E) = \mathcal{E}', l{:}\varrho \qquad \mathcal{E}'' = \mathcal{E}', l{:}(\varrho, r \mapsto C(\vec{r}))}{\mathit{extend}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}''} \text{ E-EXTEND}$$

Environment manipulation: transactions

$$\frac{\Gamma = n{:}\mathcal{E}, \Gamma' \qquad \Gamma'' = (l{:}(\mathcal{E}, l{:}\langle\rangle)), \Gamma}{\mathit{start}(l, n, \Gamma) = \Gamma''}\ \textsc{E-Start}$$

$$\frac{}{\mathit{commit}(\langle\rangle, \langle\rangle, \Gamma) = \Gamma}\ \textsc{E-Commit}_1$$

$$\frac{\begin{array}{c}\mathcal{E} = l_L{:}\varrho_L, \mathcal{E}' \qquad \varrho'_L = \mathit{release}(l(\mathcal{E}), \varrho_L) \qquad \mathcal{E}'' = l_L{:}\varrho'_L, \mathcal{E}' \\ \mathit{reflect}(n, (\mathcal{E}'', l'{:}\varrho''', \varrho''), \Gamma) = \Gamma' \qquad \mathit{commit}(\vec{n}, \vec{\mathcal{E}}, \Gamma') = \Gamma''\end{array}}{\mathit{commit}(n\ \vec{n}, \mathcal{E}\ \vec{\mathcal{E}}, \Gamma) = \Gamma''}\ \textsc{E-Commit}_2$$

Further development in the paper

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

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[5] via `async`
- protection by default
- "fragmentation" by user-command `unprotected`/"yield"
- cf. `suspend`-command in Creol

[5] of course

AME syntax

$$
\begin{array}{rcll}
v & ::= & c \mid x \mid \lambda x.e & \\
c & ::= & \text{unit} \mid \text{false} \mid \text{true} & \\
e & ::= & v & \text{expressions: values} \\
& \mid & e\ e & \text{application} \\
& \mid & \text{ref}\, e \mid\ !e \mid e := e & \\
& \mid & \text{async}\, e & \\
& \mid & \text{blockuntil}\, e & \\
& \mid & \text{unprotected}\, e &
\end{array}
$$

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 σ
2. pool of expr's/threads T
3. active expression e

Evaluation contexts

$$
\begin{array}{lcl}
\mathcal{P} & ::= & [] \mid \mathcal{P}\ e \mid \mathsf{ref}\,\mathcal{P} \mid \mathcal{P} := e \mid r := \mathcal{P} \mid \mathsf{blockuntil}\,P \\
\mathcal{U} & ::= & \mathsf{unprotected}\,\mathcal{E} \mid \mathcal{U}\ e \mid v\,\mathcal{U} \mid \mathsf{ref}\,U \mid !\mathcal{U} \mid \mathcal{U} := e \mid r := \mathcal{U} \mid \mathsf{blocku} \\
\mathcal{E} & ::= & [] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathsf{ref}\,\mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \mid \mathsf{blockuntil}\,\mathcal{E} \mid \mathsf{unprote} \\
\mathcal{F} & ::= & T.\mathcal{U}.T', \mathsf{unit} \mid T, \mathcal{P}
\end{array}
$$

$$\langle \sigma, \mathcal{F}[(\lambda x.e)\ v] \rangle \twoheadrightarrow \langle \sigma, \mathcal{F}[e[v/x]] \rangle \qquad \text{T-APP}$$

$$\frac{r \text{ fresh}}{\langle \sigma, \mathcal{F}\text{ref } v \rangle \twoheadrightarrow \langle \sigma[r \mapsto v], \mathcal{F} r \rangle} \text{ T-REF}$$

$$\frac{\sigma(r) = v}{\langle \sigma, \mathcal{F}!r \rangle \twoheadrightarrow \langle \sigma, \mathcal{F} v \rangle} \text{ T-DEREF}$$

$$\langle \sigma, \mathcal{F} r := v \rangle \twoheadrightarrow \langle \sigma[r \mapsto v], \mathcal{F}\text{unit} \rangle \qquad \text{T-SET}$$

$$\langle \sigma, \mathcal{F}\text{async } e \rangle \twoheadrightarrow \langle \sigma, e.\mathcal{F}\text{unit} \rangle \qquad \text{T-ASYNC}$$

$$\langle \sigma, \mathcal{F}\text{blockuntil true} \rangle \twoheadrightarrow \langle \sigma, \mathcal{F}\text{unit} \rangle \qquad \text{T-BOCK}$$

$$\langle \sigma, T, \mathcal{P}[] \rangle \twoheadrightarrow \langle \sigma, T.\mathcal{P}[\text{unprotected } e], \text{unit} \rangle \qquad \text{T-UNPROTECT}$$

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

$$\langle \sigma, T.e.T', \text{unit} \rangle \twoheadrightarrow \langle \sigma, T.T', e \rangle \qquad \text{T-ACTIVATE}$$

example: yielding

$$\text{yield} \triangleq \text{unprotected}\,\text{unit}$$

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

```
r1 := x;
r2 := x;
```

```
unprotected {
  x := 1
}
```

```
// A1
r1 := u
r2 := v
if (r1 != r2) {
    x := 42;
}
```

```
// A2
u++;
v++;
```

```
// U1
unprotected {
    r1 := x;
}
```

is there a race?

```
// A1
r1 := u
r2 := v
if (r1 != r2) {
    x := 42;
}
```

```
// A2
u++;
v++;
```

```
// U1
unprotected {
    r1 := x;
}
```

is there a race?

- intuitively: no race

Results

- weak = strong semantics, under certain restrictions
- violation-freedom, separation
- generalization of race-freedom [6]
- type and effect system for separation

[6] race freedom is not enough

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?
- [Grossman, 1997]
- [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 I

[Abadi et al., 2008] Abadi, M., Birell, A., Harris, T., and Isard, M. (2008).
Semantics of transactional memory and automatic mutual exclusion.
In *Proceedings of POPL '08*. ACM.

[Blundell et al., 2006] Blundell, C., Lewis, E. C., and Martin, M. K. (2006).
Subtelties of transactional memory atomicity semantics.
IEEE Computer Architecture Letters, 5(2).

[Carlstrom et al., 2006] Carlstrom, B. D., McDonald, A., Chafi, H., Chung, J., Minh, C. C., Kozyrakis, C., and Oluktun, K. (2006).
The ATOMOΣ transactional programming language.
In *ACM Conference on Programming Language Design and Implementation (Ottawa, Ontario, Canada)*. ACM.

[Garthwaite and Nettles, 1996] Garthwaite, A. and Nettles, S. (1996).
Transactions for Java.
In Aktinson, M. P. and Jordan, M. J., editors, *Proceedings of the First International Workshop on Persistence and Java. Sun Microsystems Laboratoris Technical Report 96-58*, pages 6–14.

[Grossman, 1997] Grossman, D. (1997).
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*.

[Herlihy and Wing, 1990] Herlihy, M. and Wing, J. (1990).
Linearizability: A Correctness Condition for Concurrent Objects.
ACM Transactions on Programming Languages and Systems, 12(3):463–492.

[Isard and Birell, 2007] Isard, M. and Birell, A. (2007).
Automatic mutual exclusion.
In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*.

References II

[Jagannathan et al., 2005] Jagannathan, S., Vitek, J., Welc, A., and Hosking, A. (2005).
A transactional object calculus.
Science of Computer Programming, 57(2):164–186.

[Lipton, 1975] Lipton, R. (1975).
Reduction: A method of proving properties of parallel programs.
Communications of the ACM, 18(12):717–721.
Papers from the Second ACM Symposium on POPL, Palo Alto, California.

[Shavit and Toitu, 1995] Shavit, N. and Toitu, D. (1995).
Software transactional memory.
In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 204–213.

[Welc et al., 2005] Welc, A., Jagannathan, S., and Hosking, A. (2005).
Safe futures in Java.
In *Twentieth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '05*, pages 439 – 453. ACM.
In *SIGPLAN Notices*.