An Opaque Model for Software Transactional Memory for Haskell

Rodrigo Ribeiro [1] André Du Bois [2]

DECOM, Universidade Federal de Ouro Preto (UFOP), Ouro Preto

CDTec, Universidade Federal de Pelotas(UFPel), Pelotas

2 de maio de 2017

Summary

Brief Bio

Introduction

- Motivation
- Software Transactional Memory in Haskell
- Transactional Memory Correctness

Formal semantics for transactional memory

- Core language definition
- Definitions and notations used

TL2 semantics

Checking Opacity

Conclusions

Brief Bio — (I)

- MSc in Computer Science (UFMG) - 2007;
- DSc in Computer Science (UFMG) - 2013;
- Lecturer at Universidade Federal de Ouro Preto.
- Post-doc (UFPel) - 2016 - present .

Brief Bio — (II)

- Main research topic: Type theory.
- Applications:
  - Programming languages design
  - Formal verification — proof assistants.

Currently working in... — (III)

- Formal semantics for STM.
- Formal verification of Featherweight Java typechecker.
- Formal semantics and verification of DSLs for temporal media (music, video, etc).
- Other projects
  - Formal verification of parsing algorithms.
  - Compilation of partial C programs.

This Talk — (IV)

- Formal semantics for STM.
  - Needed to ensure safety properties of STM.
  - Results reported in a paper submitted to Haskell Symposium 2017.
  - We assume that audience is familiar with Haskell syntax and monadic programming.

Multicores are coming! — (V)

- For 50 years, hardware designers delivered 40-50% increases per year in sequential program performance.
- Around 2004, this promise fails because power and cooling issues made impossible to increase clock frequencies.
- Result: If we want better performance, parallelism is no longer an option!

Parallelism and multicores — (VI)

- Parallelism is essential to improve performance on a multi-core machine.
- Unfortunately, parallel programming is immensely more error-prone than traditional sequential programming...
- How to do it? Locks and conditional variables.

Locks and programming — (VII)

- **Correct** use of locks can solve concurrency problems, but...
  - Locks are amazingly hard to use correctly!
  - Incorrect use of locks can cause races and deadlocks which are difficult to debug.
- Better solution: Transactions!

Transactional Memory — (VIII)

- Intuitivelly, write sequential code and wrap it using an **atomic** block.
- Atomic blocks execute in isolation.
  - with automatic retry if another conflicting atomic block interferes.

How does it work? — (IX)

- The TM runtime tries to interleave transactions.
- How? Here's one way:
  - Read and writes happens in a transaction local log.
  - At the end, transactions validate its log.
  - If validation succeeds, changes are committed to memory
  - If fails, re-runs from the beginning, discarding changes.

Our focus: TL2 algorithm — (X)

- Uses logs and a global clock.
- Every transaction holds its read stamp.
  - Global clock value when transaction started.
- Every transactional variable has a write stamp.
  - When reading, if write stamp $>$ read stamp - abort.

Why Haskell? — (XI)

- Haskell is pure functional language, i.e., by default, side-effects are available only throught monads.
- This allows for a strict separation between effectful and pure code.
- Using monads we can isolate transactional code from other effects present in program.

Concurrency in Haskell — (XII)

- Function fork spawns a new thread.
- It takes an action as its argument.

```
fork :: IO a → IO ThreadId
main = do {
   fork someAction;
   anotherAction;
   ...}
```

Atomic blocks in Haskell — (XIII)

- Idea: use a function atomic that guarantees atomic execution of its argument computation atomically.

```
main = do {
    r ← new 0;
    fork (atomic (someAction));
    atomic (anotherAction); ...}
```

STM Haskell interface — (XIV)

- Transactional variables:
  - **data** TVar $a = ...$
- Transactional memory monad:
  - **data** STM $a = ...$
- Creating/ reading / writing variables.

$$\text{newTVar} :: a \rightarrow \text{STM}\ (\text{TVar}\ a)$$
$$\text{readTVar} :: \text{TVar}\ a \rightarrow \text{STM}\ a$$
$$\text{writeTVar} :: \text{TVar}\ a \rightarrow a \rightarrow \text{STM}\ ()$$

STM Haskell interface — (XV)

- User controlled abort of transactions.

  $$\mathsf{retry} :: \mathsf{STM}\ a$$

- Choice operator.

  $$\mathsf{orElse} :: \mathsf{STM}\ a \rightarrow \mathsf{STM}\ a \rightarrow \mathsf{STM}\ a$$

- Running a transaction.

  $$\mathsf{atomically} :: \mathsf{STM}\ a \rightarrow \mathsf{IO}\ a$$

STM Haskell example — (XVI)

```
type Var = TVar Float
transferMoney :: Float → Var → Var → STM ()
transferMoney amount acc1 acc2
    = do
        v ← readTVar acc1
      if v ⩾ amount
        then do
          writeTVar acc1 (v − amount)
          v2 ← readTVar acc2
          writeTVar acc2 (v2 + amount)
        else retry
```

Transactional memory summary — (XVII)

- TM provides a simple programming model to write concurrent code.
- STM simplicity (for programmer) has a price:
  - Designing efficient algorithms for TM is an art.
  - Implementations usually use subtle, but efficient, algorithms.
  - How can we guarantee safety of an TM implementation?
  - What means "correctness" for TM?

Correctness criteria for STM — (XVIII)

- Several criteria proposed in literature.
- They are based on the concept of TM **history**.
- History:
  - Sequence of operations executed during a TM run.
- Example:

$$H = \left| \begin{array}{c} T_1.read\, x \rightarrow 0, \\ T_1.write(x, 1), \\ T_2.read(x) \rightarrow 1 \\ T_1.commit \\ T_2.abort \end{array} \right|$$

Correctness criteria for STM — (XIX)

- How histories can be used to stabish correctness?
- A correct history should be such that:
  - Commited transactions should appear to be executed with no other transactions.
  - Aborted transactions should leave no effect.
  - Possible to find a history H' s.t. there's a **total order** in which transactions appear to be executed.

Correctness criteria for STM — (XX)

- How histories can be used to stabish correctness?
- A correct history should be such that:
  - Possible to find a history H' s.t. there's a **total order** in which transactions appear to be executed.
- Here we'll focus on a criteria named **opacity**.
- A TM algorithm is correct if all histories generated by it are opaque.

A glimpsy of our proposal — (XXI)

- Define a TM algorithm as a small-step operational semantics over a core language.
- Semantics should produce the TM history.
- Verify that all produced histories satisfy the correctness criteria.
  - We use property based testing for this.

Core language — (XXII)

```
- values
newtype Val = Val {unVal :: Int}
- variables
newtype Var = Var {unVar :: Int}
- transaction identifiers
newtype Id = Id {unId :: Int}
- clock values
newtype Stamp = Stamp {unStamp :: Int}
```

Core language — (XXIII)

```
data Tran =
TVal Val                    - values
 | TRead Var                - read op.
 | TWrite Var Tran          - write op.
 | Tran ⊕T Tran             - composition
 | TIf Tran Tran Tran       - conditional
 | TOrElse Tran Tran        - orElse op.
 | TRetry                   - user abort
 | TAbort                   - runtime abort
```

Core language — (XXIV)

```
- transaction id + global clock info.
type TIdent = Maybe (Id, Stamp)
data Proc =
PVal Val                    - values
 | PFork Proc               - forking
 | PAtomic TIdent Tran      - atomic block
 | Proc ⊕P Proc             - composition
```

Transactional histories — (XXV)

```
data Event =
IRead Id Var Val- reading a value
 | IWrite Id Var Val- writing a value
 | IBegin Id      - beginning a transaction.
 | ICommit Id     - commit a transaction.
 | IAbort Id      - runtime abort.
 | IRetry Id      - user abort.
type History = [Event]
```

Notations: Finite maps — (XXVI)

- Empty finite map $\bullet$.
- Lookup:

$$h(x) = \begin{cases} v & \text{if } [x \mapsto v] \in h \\ \bot & \text{otherwise} \end{cases}$$

- Right biased union: $h \uplus h'$.
- Removing: $h \mid_x = h - [x \mapsto v]$, for some v.

Notations: TM state — (XXVII)

- Heaps, read and write sets represented as finite maps.
- $\Theta = \langle \Theta_h, \Theta_r, \Theta_w, \Theta_T \rangle$, where:
  - Θ_h : heap
  - Θ_r and Θ_w: finite maps between trans. id and its read / write sets.
  - Θ_T: finite map between trans. id and the transaction itself.

Reading a variable: intuition — (XXVIII)

- $\Theta(x, i, j)$: reads the value of variable x in transaction j which has read stamp i.
- How reading proceeds:
  - If x is in j write set, return its value.
  - If x is in j read set and i isn't less than x write stamp, return x associated value.
  - Lookup value in heap. If x value has a stamp greather than i, abort.
  - Otherwise, update read-set.

Reading a variable, formally — (XXIX)

$$\Theta(x,i,j) = \begin{cases} (v,\Theta) & \text{if } \Theta_w(x,j)=(i',v) \\ (v,\Theta) & \text{if } \Theta_w(x,j)=\bot, \\ & \quad \Theta_r(x,j)=(i',v), i \geqslant i' \\ (v,\Theta_r\,[j,x \mapsto v]) & \text{if } \Theta_w(x)=\Theta_r(x)=\bot, \\ & \quad \Theta_h(x)=(i',v), \text{ and } i \geqslant i' \\ \text{TAbort} & \text{if } \Theta_h(x)=(i',v) \text{ and } i<i' \end{cases}$$

Consistency check — (XXX)

- $\mathit{consistent}(\Theta, i, j)$ holds if transaction j has finished its execution on a valid view of memory.

$$\begin{aligned} \mathit{consistent}(\Theta, i, j) \;=\; & \forall x.\Theta_r(x, j) = (i, v) \rightarrow \\ & \Theta_h(x) = (i', v) \wedge i \geqslant i' \end{aligned}$$

Semantics — (XXXI)

- Transaction semantics uses triples $\langle \Theta, \sigma, t \rangle$:
  - Θ: heap and TM logs.
  - σ: TM execution history
  - t: transaction being executed.
  - i: current read stamp
  - j: current transation id.

$$\langle \Theta,\ \sigma,\ t \rangle \mapsto_{T_{ij}} \langle \Theta',\ \sigma',\ t' \rangle$$

Semantics — (XXXII)

- Process semantics uses 5-uples
  - Θ: heap and TM logs.
  - σ: TM execution history
  - i: global clock
  - j: last transaction id used.
  - p: process being executed.

$$\langle \Theta, \sigma, j, i, t \rangle \mapsto_P \langle \Theta', \sigma', j', i', t' \rangle$$

Evaluation Contexts — (XXXIII)

$$
\begin{array}{rcl}
\mathbb{T}[\cdot] & ::= & \mathsf{TWrite}\ v\ \mathbb{T}[\cdot] \\
& | & \mathbb{T}[\cdot]\ \oplus_{\mathsf{T}}\ t \\
& | & \mathsf{TVal}\ v \oplus_{\mathsf{T}}\ \mathbb{T}[\cdot] \\
& | & \mathsf{TIf}\ \mathbb{T}[\cdot]\ t\ t' \\
& | & \mathsf{TOrElse}\ \mathbb{T}[\cdot]\ t
\end{array}
$$

$$
\begin{array}{rcl}
\mathbb{P}[\cdot] & ::= & \mathsf{PFork}\ \mathbb{P}[\cdot] \\
& | & \mathbb{P}[\cdot]\ \oplus_{\mathsf{P}}\ t \\
& | & \mathsf{PVal}\ v\ \oplus_{\mathsf{P}}\ \mathbb{P}[\cdot] \\
& | & \mathsf{PAtomic}\ \mathbb{P}[\cdot]
\end{array}
$$

Evaluation context reduction — (XXXIV)

$$\frac{\langle \Theta,\ \sigma,\ t \rangle \mapsto_{T_{i\,j}} \langle \Theta',\ \sigma',\ t' \rangle}{\langle \Theta,\ \sigma,\ \mathbb{T}[t] \rangle \mapsto_{T_{i\,j}} \langle \Theta',\ \sigma',\ \mathbb{T}[t'] \rangle}\ (\mathit{TContext})$$

$$\frac{\langle \Theta,\ \sigma,\ j,\ i,\ t \rangle \mapsto_{P} \langle \Theta',\ \sigma',\ j',\ i',\ t' \rangle}{\langle \Theta,\ \sigma,\ j,\ i, \mathbb{P}[p] : s \rangle \mapsto_{P} \langle \Theta,\ \sigma',\ j',\ i',\ \mathbb{P}[p'] : s \rangle}\ (\mathit{PContext})$$

Read Semantics — (XXXV)

$$\frac{\Theta(v, i, j) = (val, \Theta') \quad \sigma' = \mathsf{IRead}\ j\ v\ val\ :\ \sigma}{\langle \Theta,\ \sigma,\ \mathsf{TRead}\ v \rangle \mapsto_{T_{ij}} \langle \Theta',\ \sigma',\ \mathsf{TVal}\ val \rangle}\ (\mathit{TReadOk})$$

$$\frac{\Theta(v, i, j) = (\mathsf{TAbort}, \Theta') \quad \sigma' = \mathsf{IAbort}\ j\ :\ \sigma}{\langle \Theta,\ \sigma,\ \mathsf{TRead}\ v \rangle \mapsto_{T_{ij}} \langle \Theta,\ \sigma',\ \mathsf{TAbort} \rangle}\ (\mathit{TReadFail})$$

Write Semantics — (XXXVI)

$$\frac{\begin{array}{c} t = \mathsf{TVal}\ val \\ \sigma' = \mathsf{IWrite}\ j\ v\ val\ :\sigma \\ \Theta' = \langle \Theta_h, \Theta_r, \Theta_w[j, x \mapsto val], \Theta_T \rangle \end{array}}{\langle \Theta,\ \sigma,\ \mathsf{TWrite}\ v\ t \rangle \mapsto_{T_{ij}} \langle \Theta',\ \sigma',\ t \rangle}\ (\mathit{TWriteVal})$$

$$\frac{}{\langle \Theta,\ \sigma,\ \mathsf{TWrite}\ v\ \mathsf{TFail} \rangle \mapsto_{T_{ij}} \langle \Theta,\ \sigma,\ \mathsf{TFail} \rangle}\ (\mathit{TWriteFail})$$

Composition Semantics — (XXXVII)

$$\frac{val = val_1 + val_2}{\langle \Theta, \sigma, (\mathsf{TVal}\ val_1) \oplus_{\mathsf{T}} (\mathsf{TVal}\ val_2) \rangle \mapsto_{T_{i\,j}} \langle \Theta, \sigma, \mathsf{TVal}\ val \rangle} \ (TAddVal)$$

$$\frac{}{\langle \Theta, \sigma, \mathsf{TFail} \oplus_{\mathsf{T}} t \rangle \mapsto_{T_{i\,j}} \langle \Theta', \sigma', \mathsf{TFail} \rangle} \ (TAddL)$$

$$\frac{}{\langle \Theta, \sigma, (\mathsf{TVal}\ val) \oplus_{\mathsf{T}} \mathsf{TFail} \rangle \mapsto_{T_{i\,j}} \langle \Theta', \sigma', \mathsf{TFail} \rangle} \ (TAddR)$$

Conditional Semantics — (XXXVIII)

$$\frac{}{\langle \Theta, \sigma, \mathsf{TIf}\ (\mathsf{TVal}\ 0)\ t\ t' \rangle \mapsto_{T_{ij}} \langle \Theta, \sigma,\ t \rangle}\ (\mathit{TIfZero})$$

$$\frac{v \neq 0}{\langle \Theta, \sigma, \mathsf{TIf}\ (\mathsf{TVal}\ v)\ t\ t' \rangle \mapsto_{T_{ij}} \langle \Theta, \sigma,\ t' \rangle}\ (\mathit{TIfNonZero})$$

$$\frac{}{\langle \Theta,\ \sigma,\ \mathsf{TIf}\ \mathsf{TFail}\ t\ t' \rangle \mapsto_{T_{ij}} \langle \Theta,\ \sigma,\ \mathsf{TFail} \rangle}\ (\mathit{TIfFail})$$

OrElse Semantics — (XXXIX)

$$\frac{}{\langle \Theta, \sigma, \mathsf{TOrElse}\,(\mathsf{TVal}\ v)\ t' \rangle \mapsto_{T_{ij}} \langle \Theta', \sigma', \mathsf{TVal}\ v \rangle}\ (\mathit{TOrElseVal})$$

$$\frac{}{\langle \Theta, \sigma, \mathsf{TOrElse}\ \mathsf{TRetry}\ t' \rangle \mapsto_{T_{ij}} \langle \Theta, \sigma, t' \rangle}\ (\mathit{TOrElseR})$$

$$\frac{}{\langle \Theta, \sigma, \mathsf{TOrElse}\ \mathsf{TAbort}\ t' \rangle \mapsto_{T_{ij}} \langle \Theta, \sigma, \mathsf{TAbort} \rangle}\ (\mathit{TOrElseA})$$

Process preemption — (XL)

$$\frac{\langle \Theta,\ \sigma,\ j,\ i,\ s \rangle \mapsto_P \langle \Theta',\ \sigma',\ j',\ i',\ s' \rangle}{\langle \Theta,\ \sigma,\ j,\ i,\ p : s \rangle \mapsto_P \langle \Theta',\ \sigma',\ j',\ i',\ p : s' \rangle}\ (PPreempt)$$

Forking semantics — (XLI)

$$\frac{s' = \mathsf{PVal}\,0 : p : s}{\langle \Theta,\ \sigma,\ j,\ i,\ (\mathsf{PFork}\,p) : s \rangle \mapsto_P \langle \Theta,\ \sigma,\ j,\ i,\ s' \rangle}\ (\mathit{PFork})$$

Atomic block semantics — (XLII)

$$\sigma' = \mathsf{IBegin}\ j{:}\sigma$$

$$s' = s +\!\!+ [\mathsf{PAtomic}\ (i,j)\ t]$$

$$\Theta_1 = \langle \Theta_h, \Theta_r\,[j \mapsto \bullet], \Theta_w\,[j \mapsto \bullet], \Theta_T\,[j \mapsto t] \rangle$$

$$\frac{}{\langle \Theta, \sigma, j, i, \mathsf{PAtomic}\ ()\ t{:}s \rangle \mapsto_P \langle \Theta', \sigma', j{+}1, i, s' \rangle}\ (PAt1)$$

Atomic block semantics — (XLIII)

$$\frac{v{=}\mathsf{TVal}\ n \qquad \mathit{consistent}(\Theta,i,j) \qquad \sigma'{=}\mathsf{ICommit}\ j{:}\sigma \qquad \Theta'{=}\langle \Theta'_h, \Theta_r|_j, \Theta_w|_j\rangle \qquad \Theta'_h{=}\Theta_h \uplus \Theta_w(j)}{\langle \Theta, \sigma, j, i, \mathsf{PAtomic}\ v{:}s\rangle \mapsto_P \langle \Theta', \sigma', j, i{+}1, \mathsf{PVal}\ n{:}s\rangle}\ (\mathit{PAt2})$$

Atomic block semantics — (XLIV)

$$\frac{\Theta' = \langle \Theta_h, \Theta_r|_j, \Theta_w|_j, \Theta_T|_j \rangle \quad s' = s +\!\!+ \mathsf{PAtomic}\,()\,t \quad t = \Theta_t(j)}{\langle \Theta, \sigma, j, i, \mathsf{PAtomic\ TFail}{:}s \rangle \mapsto_P \langle \Theta', \sigma, j{+}1, i, s' \rangle}\ (PAt3)$$

Checking Opacity — (XLV)

- We check safety properties using Quickcheck, a Haskell library for property based testing.
- Library formed by combinators for building generators and properties.
  - Generators build random values which are checked against relevant properties.
  - Combinators for properties mimics first-order logic conectives and quantifiers.

Defining Opacity — (XLVI)

- A TM algorithm is opaque if all prefixes of its generated histories are final state opaque.
- In Haskell:

$$\begin{aligned} &\mathsf{opacity} :: \mathsf{History} \to \mathsf{Bool} \\ &\mathsf{opacity} = \mathsf{all}\ \mathsf{finalStateOpacity} \circ \mathsf{inits} \end{aligned}$$

Defining Opacity — (XLVII)

- A history is final state opaque if exists some completion of it that preserves real time order and all of its transactions are legal.
- In Haskell:

```
finalStateOpacity :: History → Bool
finalStateOpacity
    = some prop ∘ completions
      where
        prop tr = preservesOrder tr ∧ legal tr
        some p xs = (null xs) ∨ (any p xs)
```

Defining Opacity — (XLVIII)

- Completion of a history H is a history S, s.t.
  - All live and non-commit pending transactions of H are aborted in S; and
  - All commit pending transactions of H are aborted or committed in S.
- Our model we do not consider commit-pending transactions.
- Complete a history is just abort all live transactions.

Defining Opacity — (XLIX)

- Real time order: $t_k \prec_H t_m$
  - t_k is completed and its last event occurs before t_ms first.
- A history H' preserves the real time ordering of H:

$$\forall t_k\ t_m . t_k \prec_H t_m \rightarrow t_k \prec_{H'} t_m$$

Defining Opacity — (L)

- Legal histories.
  - A history is said to be *legal* if all values read after a write in a variable are equal to last value written

Checking Opacity — (LI)

- We use function opacity as a Quickcheck property over histories produced by executing random generated programs.
- Semantics passes in all test-suit runs.

Checking Opacity — (LII)

- So, our semantics does enjoy opacity?
  - As pointed by Djikstra: “Tests can only prove the presence of bugs...”.
- In order to afirm this categorically, we need a formal proof.
  - A venue that we intend to pursue in future work.

History not told... (LIII)

- Alternative STM-Haskell based semantics (does not have opacity).
- Compilation to virtual machine.
- Another correctness criteria: markability.
- Other properties verified
  - Compilation preserves safety properties.
  - Equivalence of safety properties.

Conclusions — (LIV)

- We define small-step operational semantics for STM-Haskell like language.
  - TL2-based semantics.
  - STM-Haskell based semantics.
- Implemented interpreters for the semantics and relevant properties over it using Haskell.

Conclusions — (LV)

- Property based testing.
  - Allows for quickly identify errors in semantics — counter-examples.
  - Provides some degree of assurance, before using more formal approachs.