Under consideration for publication in J. Functional Programming

Towards a Verified Opaque Semantics for Software Transactional Memory

RODRIGO RIBEIRO
Universidade Federal de Ouro Preto, Minas Gerais, Brazil

ANDRÉ RAUBER DU BOIS
Universidade Federal de Pelotas, Rio Grande do Sul, Brazil
(*e-mail:* rodrigo@decsi.ufop.br)

Abstract

Here goes the abstract...

1 Introduction

It is a well known fact that writing correct lock based concurrent programs is a painful task, since locks doesn't scale (Harris *et al.*, 2005). *Software Transactional Memory* (STM) is a promising approach to write concurrent software since it can perform groups of memory operations atomically (Shavit & Touitou, 1995) thus providing automatic rollback, deadlock, priority inversion and lock granularity freedom.

Haskell's implementation of STM pioneered, thought its type system, the strict separation between transactional and non-transactional code and its interface allows the compositional definition of transactions (Harris *et al.*, 2005).

Talk about models / semantics of STM. Mention Hutton work.

More specifically, our contributions are:

- We define a trace-based small step semantics for a high-level language with STM support.
- A Haskell implementation of opacity property for traces generated by proposed language semantics and its test using QuickCheck (Claessen & Hughes, 2000) and HPC (Gill & Runciman, 2007).

Our work shares some similarities with (**?**): We use a small variation of its reduced language for STM, but our focus is on verify a safety property (namely, opacity ()) for STM, instead of a compiler correctness theorem for transactional virtual machine.

2 A Model for Software Transactional Memory

In order to analyse a semantics for a high level language with transactional memory support, we follow the same approach used by (Hu & Hutton, 2009): defining a simple language of integer-valued expressions (e.g. Figure 1) extended with conditionals, *orElse* and *retry* constructs.

We let meta-variable v denote integer constants and x (transactional) variables.

$$
\begin{array}{rcl}
p & ::= & v \\
& | & \mathit{fork}\ p \\
& | & \mathit{Atomic}\ t \\
& | & p \oplus p
\end{array}
\qquad
\begin{array}{rcl}
t & ::= & v \\
& | & \mathit{readTVar}\ x \\
& | & \mathit{writeTVar}\ x\ t \\
& | & t \oplus t \\
& | & \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ t \\
& | & \mathit{retry} \\
& | & t\ \mathit{orElse}\ t
\end{array}
$$

Fig. 1. Language syntax.

In our semantics, we use finite maps to represent heaps and logs used by transactions (i.e. read and write sets). Notation $\bullet$ denotes an empty finite mapping. Variable Θ represents a triple formed by a heap, read and write set used by some transaction t. We represent read and write sets and the heap by Θ_r, Θ_w and Θ_h respectively. Let $h(x)$ denotes the operation of retrieving the value associated with key x in finite mapping h and $h(x) = \perp$ denotes that no value is associated with x. Notation $h \uplus h'$ denotes the right-biased union of two finite mappings, i.e. when both maps have the same key x, we keep the value $h'(x)$. Following common practice, notation $h\,[x \mapsto v]$ denotes finite mapping update, i.e. finite mapping h' such that: 1) $h'(x) = v$ and 2) $h'(y) = h(y)$, for $x \neq y$.

The proposed semantics is based on Transactional Locking 2 algorithm (TL2) (Dice *et al.*, 2006). We define it using a small-step semantics defined in Figures 5 and 7. We use evaluation contexts, which are defined in Figure 2, to avoid the need of “congruence” rules in semantics.

Evaluation contexts for transactions

$$
\begin{array}{rcl}
\mathbb{T}[\cdot] & ::= & \mathit{writeTVar}\ x\ \mathbb{T}[\cdot] \\
& | & \mathbb{T}[\cdot] \oplus t \\
& | & v \oplus \mathbb{T}[\cdot] \\
& | & \mathit{if}\ \mathbb{T}[\cdot]\ \mathit{then}\ t\ \mathit{else}\ t'
\end{array}
$$

Evaluation contexts for processes

$$
\begin{array}{rcl}
\mathbb{P}[\cdot] & ::= & \mathit{fork}\ \mathbb{P}[\cdot] \\
& | & \mathbb{P}[\cdot] \oplus t \\
& | & t \oplus \mathbb{P}[\cdot]
\end{array}
$$

Fig. 2. Evaluation contexts for high-level language.

Informally, TL2 algorithm works as follows: threads execute reads and writes to objects, but no memory locations are actually modified. All writes and reads are recorded in write and read logs. When a transaction finishes, it validates its log to check if it has seen a consistent view of memory, and its changes are committed to memory.

$$\Theta(x,i) = \begin{cases} (v,\Theta) & \textit{if } \Theta_w(x) = v \\ (v,\Theta) & \textit{if } \Theta_w(x) = \bot, \Theta_r(x) \neq \bot, \Theta_h(x) = (i',v) \textit{ and } i \geq i' \\ (v,\Theta_r[x \mapsto v]) & \textit{if } \Theta_w(x) = \Theta_r(x) = \bot,\ \Theta_h(x) = (i',v),\ \textit{and } i \geq i' \\ \textit{retry} & \textit{if } \Theta_h(x) = (i',v) \textit{ and } i < i' \end{cases}$$

Fig. 3. Reading a variable

$$\textit{consistent}(\Theta,i) = \forall x.\Theta_r(x) = (i,v) \to \Theta_h(x) = (i',v) \wedge i \geq i'$$

Fig. 4. Predicate for consistency of transaction logs

Verifying consistency of a transaction.

Now the small step semantics for processes

References

Claessen, Koen, & Hughes, John. (2000). Quickcheck: A lightweight tool for random testing of haskell programs. *Pages 268–279 of: Proceedings of the fifth acm sigplan international conference on functional programming*. ICFP '00. New York, NY, USA: ACM.

Dice, Dave, Shalev, Ori, & Shavit, Nir. (2006). Transactional locking ii. *Pages 194–208 of: Proceedings of the 20th international conference on distributed computing*. DISC'06. Berlin, Heidelberg: Springer-Verlag.

Gill, Andy, & Runciman, Colin. (2007). Haskell program coverage. *Pages 1–12 of: Proceedings of the acm sigplan workshop on haskell workshop*. Haskell '07. New York, NY, USA: ACM.

Harris, Tim, Marlow, Simon, Peyton-Jones, Simon, & Herlihy, Maurice. (2005). Composable memory transactions. *Pages 48–60 of: Proceedings of the tenth acm sigplan symposium on principles and practice of parallel programming*. PPoPP '05. New York, NY, USA: ACM.

Hu, Liyang, & Hutton, Graham. (2009). Towards a Verified Implementation of Software Transactional Memory. Achten, Peter, Koopman, Pieter, & Morazan, Marco (eds), *Trends in Functional Programming volume 9*. Intellect. Selected papers from the Ninth Symposium on Trends in Functional Programming, Nijmegen, The Netherlands, May 2008.

Shavit, Nir, & Touitou, Dan. (1995). Software transactional memory. *Pages 204–213 of: Proceedings of the fourteenth annual acm symposium on principles of distributed computing*. PODC '95. New York, NY, USA: ACM.

$$\boxed{\langle \Theta, \sigma, t \rangle \mapsto_{T_i} \langle \Theta', \sigma', t' \rangle}$$

$$\frac{\langle \Theta, \sigma, t \rangle \mapsto_{T_i} \langle \Theta', \sigma', t' \rangle}{\langle \Theta, \sigma, \mathbb{T}[t] \rangle \mapsto_{T_i} \langle \Theta', \sigma', \mathbb{T}[t'] \rangle} \ (Context) \qquad \frac{\sigma' = Write\ i\, x\, v :: \sigma}{\langle \Theta, \sigma, writeTVar\ x\, v \rangle \mapsto_{T_i} \langle \Theta_w[x \mapsto v], \sigma', t' \rangle} \ (Write)$$

$$\frac{\sigma' = Abort\ i :: \sigma}{\langle \Theta, \sigma, writeTVar\ x\, retry \rangle \mapsto_{T_i} \langle \Theta, \sigma', retry \rangle} \ (WriteRetry) \qquad \frac{\sigma' = Read\ i\, x\ v :: \sigma \quad (v, \Theta') = \Theta(x, i)}{\langle \Theta, \sigma, readTVar\ x \rangle \mapsto_{T_i} \langle \Theta', \sigma', v \rangle} \ (ReadVal)$$

$$\frac{\sigma' = Abort\ i :: \sigma \quad retry = \Theta(x, i)}{\langle \Theta, \sigma, readTVar\ x \rangle \mapsto_{T_i} \langle \Theta, \sigma', retry \rangle} \ (ReadRetry) \qquad \frac{}{\langle \Theta, \sigma, v \oplus v' \rangle \mapsto_{T_i} \langle \Theta, \sigma, v + v' \rangle} \ (Plus)$$

$$\frac{\langle \Theta, \sigma, t \rangle \mapsto^{\star}_{T_i} \langle \Theta', \sigma', v \rangle}{\langle \Theta, \sigma, t\ orElse\ t' \rangle \mapsto_{T_i} \langle \Theta', \sigma', v \rangle} \ (OrElseVal) \qquad \frac{\langle \Theta, \sigma, t \rangle \mapsto^{\star}_{T_i} \langle \Theta', \sigma', retry \rangle}{\langle \Theta, \sigma, t\ orElse\ t' \rangle \mapsto_{T_i} \langle \Theta', \sigma', retry \rangle} \ (OrElseRetry)$$

$$\frac{\sigma' = Abort\ i :: \sigma}{\langle \Theta, \sigma, retry \oplus t \rangle \mapsto_{T_i} \langle \Theta, \sigma', retry \rangle} \ (PlusRetryL) \qquad \frac{\sigma' = Abort\ i :: \sigma}{\langle \Theta, \sigma, t \oplus retry \rangle \mapsto_{T_i} \langle \Theta, \sigma', retry \rangle} \ (PlusRetryR)$$

$$\frac{}{\langle \Theta, \sigma, if\ v = 0\ then\ t\ else\ t' \rangle \mapsto_{T_i} \langle \Theta, \sigma, t \rangle} \ (IfZero)$$

$$\frac{}{\langle \Theta, \sigma, if\ v \neq 0\ then\ t\ else\ t' \rangle \mapsto_{T_i} \langle \Theta, \sigma, t' \rangle} \ (IfNonZero)$$

$$\frac{\sigma' = Abort\ i :: \sigma}{\langle \Theta, \sigma, if\ retry\ then\ t\ else\ t' \rangle \mapsto_{T_i} \langle \Theta, \sigma', retry \rangle} \ (IfRetry)$$

Fig. 5. Small-step operational semantics for transactions.

$$\boxed{\langle \Theta, \sigma, t \rangle \mapsto^{\star}_{T_i} \langle \Theta', \sigma', t' \rangle}$$

$$\frac{}{\langle \Theta, \sigma, t \rangle \mapsto^{\star}_{T_i} \langle \Theta, \sigma, t \rangle} \ (TRefl)$$

$$\frac{\langle \Theta, \sigma, t \rangle \mapsto_{T_i} \langle \Theta_1, \sigma_1, t_1 \rangle \quad \langle \Theta_1, \sigma_1, t_1 \rangle \mapsto^{\star}_{T_i} \langle \Theta', \sigma', t' \rangle}{\langle \Theta, \sigma, t \rangle \mapsto^{\star}_{T_i} \langle \Theta', \sigma', t' \rangle} \ (TTran)$$

Fig. 6. Reflexive-transitive closure of transaction small step semantics.

$$\boxed{\langle h,\sigma,i,s\rangle \mapsto_P \langle h',\sigma',i',s'\rangle}$$

$$\frac{\langle h,\sigma,i,s\rangle \mapsto_P \langle h',\sigma',i',s'\rangle}{\langle h,\sigma,i,p:s\rangle \mapsto_P \langle h',\sigma',i',p:s'\rangle}\ (\mathit{Preempt}) \qquad \frac{\langle h,\sigma,i,p:s\rangle \mapsto_P \langle h',\sigma',i',p':s'\rangle}{\langle h,\sigma,i,\mathbb{P}[p]:s\rangle \mapsto_P \langle h',\sigma',i,\mathbb{P}[p']:s\rangle}\ (\mathit{Context})$$

$$\frac{}{\langle h,\sigma,i,v\oplus v':s\rangle \mapsto_P \langle h,\sigma,i,v+v':s\rangle}\ (\mathit{Plus}) \qquad \frac{}{\langle h,\sigma,i,\mathit{fork}\ p:s\rangle \mapsto_P \langle h,\sigma,i,0:p:s\rangle}\ (\mathit{Fork})$$

$$\frac{\mathit{consistent}(\Theta',i) \quad h' = h \uplus \Theta_w \quad \langle (h,\bullet,\bullet),\sigma,t\rangle \mapsto^{\star}_{T_i} \langle \Theta',\sigma',v\rangle}{\langle h,\sigma,i,\mathit{Atomic}\ t:s\rangle \mapsto_P \langle h',\sigma',i+1,v:s\rangle}\ (\mathit{AtomicVal})$$

$$\frac{\langle (h,\bullet,\bullet),\sigma,t\rangle \mapsto^{\star}_{T_i} \langle \Theta',\sigma',\mathit{retry}\rangle}{\langle h,\sigma,i,\mathit{Atomic}\ t:s\rangle \mapsto_P \langle \sigma',i,h,0:s\rangle}\ (\mathit{AtomicRetry}) \qquad \frac{\langle (h,\bullet,\bullet),\sigma,t\rangle \mapsto^{\star}_{T_i} \langle \Theta',\sigma',t'\rangle \quad \neg\mathit{consistent}(\Theta',i)}{\langle h,\sigma,i,\mathit{Atomic}\ t:s\rangle \mapsto_P \langle \sigma',i,h,0:s\rangle}\ (\mathit{AtomicNon})$$

Fig. 7. Small-step operational semantics for processes.