On the Operational Behaviour of Weak Consistency Models with Atomic Visibility

Andrea Cerone[1]

[1]Imperial College London, UK, a.cerone@imperial.ac.uk

March 5, 2018

The objective of these notes is that of defining a small step operational semantics for consistency models. Judgements will define small step reductions $\mathcal{C} \xrightarrow{\lambda} \mathcal{C}'$ between the configurations $\mathcal{C}, \mathcal{C}'$. These contain the information about the state of the transactional memory, and the concurrent program to be executed. The former will be represented using abstract executions [1].

1 Computational Model

We focus on a computational model where multi-threaded programs can access and update locations in a centralised heap using atomic transactions. A heap $h \in \textsc{Heap}$ consists of a partial function from a set of addresses $a \in \textsc{Addr} \triangleq \left\{[n] \;\middle|\; n \in \mathbb{N}\right\}$ to values in $v \in \textsc{Val} \triangleq \mathbb{N} \cup \textsc{Addr}$. The set of all heaps is denoted by $\textsc{Heap}$. Each thread has its own stack, where data for performing local computations is stored. Transactions also have a transaction-local stack. The set of thread-local stack variables is denoted by $\textsc{ThdVars} \triangleq \{\mathtt{x}, \mathtt{y}, \cdots\}$, while the set of transaction local locations is denoted by $\textsc{TxVars} \triangleq \{\mathtt{a}, \mathtt{b}, \cdots\}$. We use $\textsc{ThreadStacks}$ to range over thread-local stacks in the set $\textsc{ThreadStacks} \triangleq \textsc{ThdVars} \to \textsc{Val}$, and $\textsc{TxStacks}$ to range over transaction-local stacks in the set $\textsc{TxStacks} \triangleq \textsc{TxVars} \to \textsc{Val}$.

The transaction-local stack is created at the moment a transaction start, and is destroyed at the moment it commits. Transactions can read from, but cannot write to, the thread-local stack. This assumption makes it possible to abstract from aborting transactions, as these would have no side-effects in the computational described. We assume that each transaction-local stack has a special variable that is used to store the value returned by the transaction upon commit. Each thread-local stack also comes equipped with a special return variable, where the contents of the value returned by transactions are stored. We use the symbol `ret` to denote the special return variable, both in transaction-local and thread-local stacks.

SX: Not sure about the ret var, how to transfer the ret var from tx stack to thread stack as we cannot modify the thread stack.

We leave the consistency model of the transactional memory unspecified. The rules of our operational semantics will be parametric in the specification of a consistency model, using the style of specification proposed in [2].

Syntax of Programs We assume a set of (primitive) transactional commands $t, t', \cdots$, which we leave unspecified. Each transactional command t is associated to a *state transformer* $\mathcal{S}_t \subseteq (\textsc{ThreadStacks} \times \textsc{TxStacks} \times \textsc{Heap}) \times (\textsc{TxStacks} \times \textsc{Heap})$. We use the notation $(\sigma, \tau, h) \rightsquigarrow (\tau', h')$ in lieu of $((\sigma, \tau, h), (\tau', h')) \in \mathcal{S}_t$. Note that this definition ensures that primitive transactional commands cannot update the thread-local stack. We also assume a set of primitive (non-transactional) commands $c, c', \cdots$ that can be executed by a command outside transactions. Each primitive non-transactional command c is associated with a state transformer $\mathcal{S}_c \subseteq (\textsc{ThreadStacks} \times \textsc{ThreadStacks})$, and again we adopt the notation $\sigma \overset{\sigma}{\rightsquigarrow}_c {}'$ in lieu of $(\sigma, \sigma') \in \mathcal{S}_c$. This definition ensures that thread-local stack do not access neither the transaction-local stack, nor the heap.

Often, we will assume a language of expressions at the base of primitive (transactional and non-transactional) commands. This language is defined by the grammar below:

$$\mathbb{E} \quad ::= \quad v \mid \mathtt{x} \mid \mathsf{a} \mid \mathbb{E} + \mathbb{E} \mid \mathbb{E} \cdot \mathbb{E} \mid \cdots$$

The set of all expressions is denoted by $\textsc{Expr}$. Because non-transactional commands cannot access the thread-local stack, we will need the following, inductively defined, predicate:

$$\begin{array}{rcl} \mathsf{isThreadLocal}(v) & = & \mathtt{true} \\ \mathsf{isThreadLocal}(\mathtt{x}) & = & \mathtt{true} \\ \mathsf{isThreadLocal}(\mathsf{a}) & = & \mathtt{false} \\ \mathsf{isThreadLocal}(\mathbb{E}_1 + \mathbb{E}_2) & = & \mathsf{isThreadLocal}(\mathbb{E}_1) \land \mathsf{isThreadLocal}(\mathbb{E}_2) \\ \vdots & \vdots & \vdots \end{array}$$

In general, if the language of expressions contains an operator $\mathsf{f}\,(\mathbb{E}_1, \cdots, \mathbb{E}_n)$, we define $\mathsf{isThreadLocal}(\mathsf{f}\,(\mathbb{E}_1, \cdots, \mathbb{E}_n)) = \bigwedge_{i=1,\cdots,n} \mathsf{isThreadLocal}(\mathbb{E}_i)$.

The set of primitive commands we will use is given by

$$\begin{array}{rcl} c & ::= & \mathtt{x} := \mathbb{E} \mid \mathsf{assume}(\mathbb{E}) \\ t & ::= & \mathsf{a} := \mathbb{E} \mid [\mathbb{E}] := \mathbb{E} \mid \mathsf{a} := [\mathbb{E}] \mid \mathsf{assume}(\mathbb{E}) \end{array}$$

where in the right-hand sides of the c clause we always require that $\mathsf{isThreadLocal}(\mathbb{E}) = \mathtt{true}$.

Below we define the syntax of programs allowed by our language.

$$\begin{array}{rcl} \mathbb{P} & ::= & \mathbf{0} \mid \mathtt{tid} : \mathbb{C} \parallel \mathbb{P} \\ \mathbb{C} & ::= & \mathbf{0} \mid X \mid \pi.\mathbb{C} \mid \mathbb{C} + \mathbb{C} \mid \mu X.\mathbb{C} \\ \pi & ::= & c \mid \lceil \mathbb{T} \rceil \\ \mathbb{T} & ::= & \mathbf{0} \mid X \mid t.\mathbb{T} \mid \mathbb{T} + \mathbb{T} \mid \mu X.\mathbb{T} \end{array}$$

SX: what is the different between . and ; , *i.e.* $t \mid \mathbb{T} ; \mathbb{T} \mid \ldots$.

Note that each thread has a unique thread identifier `tid` associated. The set of all thread identifiers is `Tids`.

2 Interpretation of Expressions and Primitive Commands

Expressions are going to be evaluated in values from $\mathbb{N}$ in the usual way. Note that we need to account for the fact that we have two different notions of stacks, one transaction-local, and the other thread-local.

$$\begin{array}{rcl}
[\![\cdot]\!](.)(.) & : & \textsc{Expr} \times \textsc{ThreadStacks} \times \textsc{TxStacks} \to \mathbb{N} \\
[\![v]\!](\sigma)(\tau) & \triangleq & v \\
[\![\mathtt{x}]\!](\sigma)(\tau) & \triangleq & \sigma(\mathtt{x}) \\
[\![\mathsf{a}]\!](\sigma)(\tau) & \triangleq & \tau(\mathsf{a}) \\
[\![E_1 + E_2]\!](\sigma)(\tau) & \triangleq & [\![E_1]\!](\sigma)(\tau) + [\![E_2]\!](\sigma)(\tau) \\
\vdots & \triangleq & \vdots
\end{array}$$

Note that, for any expression $\mathbb{E}$ such that $\mathsf{isThreadLocal}(\mathbb{E}) = \mathtt{true}$, we have that $[\![\mathbb{E}]\!](\sigma)(\tau) = [\![\mathbb{E}]\!](\sigma)(\tau')$ for any $\tau, \tau' \in \textsc{TxStacks}$ and $\sigma \in \textsc{ThreadStacks}$. In this case, we commit an abuse of notation and write $[\![\mathbb{E}]\!](\sigma)$ as a shorthand for $[\![\mathbb{E}]\!](\sigma)(\tau_0)$, where $\tau_0 = \lambda\mathsf{a}.0$.

SX: why $\tau_0 = \lambda\mathsf{a}.0$ instead of any? I guess any is also ok.

We now proceed to define the state transformers associated to transactional and non-transactional primitive commands. For transactional primitive commands we have

$$\begin{array}{lcl}
(\sigma, \tau, h) & \overset{\mathsf{a}:=\mathbb{E}}{\rightsquigarrow} & (\tau[\mathsf{a} \mapsto [\![\mathbb{E}]\!](\sigma)(\tau)], h) \\
(\sigma, \tau, h) & \overset{\mathsf{a}:=[\mathbb{E}]}{\rightsquigarrow} & (\tau[\mathsf{a} \mapsto h([\![\mathbb{E}]\!](\sigma)(\tau))], h) \\
(\sigma, \tau, h) & \overset{[\mathbb{E}_1]:=\mathbb{E}_2}{\rightsquigarrow} & (\tau, h[[\![\mathbb{E}_1]\!](\sigma)(\tau) \mapsto [\![\mathbb{E}]\!](\sigma)(\tau)]) \\
(\sigma, \tau, h) & \overset{\mathsf{assume}(\mathbb{E})}{\rightsquigarrow} & (\sigma, h) \text{ where } [\![E]\!](\sigma)(\tau) \neq 0
\end{array}$$

For non-transactional primitive commands we have

$$\begin{array}{lcl}
\sigma & \overset{\mathtt{x}:=\mathbb{E}}{\rightsquigarrow_c} & \sigma[\mathtt{x} \mapsto [\![\mathbb{E}]\!](\sigma)] \text{ where } \mathsf{isThreadLocal}(\mathbb{E}) = \mathtt{true} \\
\sigma & \overset{\mathsf{assume}(E)}{\rightsquigarrow_c} & \sigma \text{ where } \mathsf{isThreadLocal}(\mathbb{E}) = \mathtt{true} \wedge [\![\mathbb{E}]\!](\sigma) \neq 0
\end{array}$$

For some transactional primitive command t, we also defined its fingerprint $\mathsf{Fprint} : \mathbb{C} \times \textsc{ThreadStacks} \times \textsc{TxStacks} \times \textsc{Heap} \to (\{\mathtt{read}, \mathtt{write}\} \times \textsc{Addr} \times \textsc{Val})$. This denotes the kind of operation that performed by t.

$$\begin{array}{rcl}
\mathsf{Fprint}(\mathsf{a} := \mathbb{E}, -, -, -) & \triangleq & \text{undefined} \\
\mathsf{Fprint}(\mathsf{a} := [\mathbb{E}], \sigma, \tau, h) & \triangleq & (\mathtt{read}, [\![\mathbb{E}]\!](\sigma)(\tau), h([\![E]\!](\sigma)(\tau)) \\
\mathsf{Fprint}([\mathbb{E}_1] := \mathbb{E}_2, \sigma, \tau, h) & \triangleq & (\mathtt{write}, [\![\mathbb{E}_1]\!](\sigma)(\tau), h([\![E_2]\!](\sigma)(\tau))
\end{array}$$

SX: Previously we are setting up the t as in a very generous form, now we are saying some of then can be extracted to a read/write form. Here it is a bit inconsistent. Minor point but might change later.
Possibly finger print heap with $\mathsf{ws}\,(h)$ and $\mathsf{rs}\,(h)$ functions is a good way to go.

3 Semantics of Transactions

The semantics of transactions is given in an operational way, judgements take the form $\sigma \vdash \langle \tau, h, \mathsf{RS}, \mathsf{WS}, \mathbb{T} \rangle \to \langle \tau', h', \mathsf{RS}', \mathsf{WS}', \mathbb{T}' \rangle$. Note that in this semantics the σ component cannot be manipulated by performing a transition, as to reflect the fact that transactions can only read from, and never write to, the thread local heap. The components $\mathsf{RS}, \mathsf{WS} \in \textsc{Addr} \rightharpoonup \textsc{Val}$ record the read-set and write-set associated to the transaction code, respectively.

In order to give the semantics of transactions, it will be useful to define the following operators over sets of key-value pairs.

$$\begin{array}{rcl} \mathsf{RS} \mathbin{\Diamond\!\!\!\!\!+} (a, v) & \triangleq & \mathsf{RS} \uplus \left\{ a \mapsto v \right\} \\ \mathsf{WS} \oplus (a, v) & \triangleq & \mathsf{WS}[a \mapsto v] \end{array}$$

The rules of the operational semantics for transactions are the following:

$$\frac{(\sigma, \tau, h) \overset{t}{\leadsto} (\tau', h') \qquad \mathsf{Fprint}(t, \sigma, \tau, h) = \text{undefined}}{\sigma \vdash \langle \tau, h, \mathsf{RS}, \mathsf{WS}, t.\mathbb{T} \rangle \to \langle \tau', h', \mathsf{RS}, \mathsf{WS}, \mathbb{T} \rangle} \; (prim{-}t{-}local)$$

$$\frac{(\sigma, \tau, h) \overset{t}{\leadsto} (\tau', h') \qquad \mathsf{Fprint}(t, \sigma, \tau, h) = (\mathtt{read}, a, v)}{\sigma \vdash \langle \tau, h, \mathsf{RS}, \mathsf{WS}, t.\mathbb{T} \rangle \to \langle \tau', h', \mathsf{RS} \mathbin{\Diamond\!\!\!\!\!+} (a, v), \mathsf{WS}, \mathbb{T} \rangle} \; (prim{-}t{-}read)$$

$$\frac{(\sigma, \tau, h) \leadsto_t (\tau', h') \qquad \mathsf{Fprint}(t, \sigma, \tau, h) = (\mathtt{write}, a, v)}{\sigma \vdash \langle \tau, h, \mathsf{RS}, \mathsf{WS}, t.\mathbb{T} \rangle \to \langle \tau', h', \mathsf{RS}, \mathsf{WS} \oplus (a, v), \mathbb{T} \rangle} \; (prim{-}t{-}write)$$

$$\frac{}{\sigma \vdash \langle \tau, h, \mathsf{RS}, \mathsf{WS}, \mathbb{T}_1 + \mathbb{T}_2 \rangle \to \langle \tau, h, \mathsf{RS}, \mathsf{WS}, \mathbb{T}_1 \rangle} \; (\mathsf{T}{-}choice{-}L)$$

$$\frac{}{\sigma \vdash \langle \tau, h, \mathsf{RS}, \mathsf{WS}, \mathbb{T}_1 + \mathbb{T}_2 \rangle \to \langle \tau, h, \mathsf{RS}, \mathsf{WS}, \mathbb{T}_2 \rangle} \; (\mathbb{T}{-}choice{-}R)$$

$$\frac{}{\sigma \vdash \langle \tau, h, \mathsf{RS}, \mathsf{WS}, \mu X.\mathbb{T} \rangle \to \langle \tau, h, \mathsf{RS}, \mathsf{WS}, \{{}^{\mu X.\mathbb{T}}/{}_{X}\}\mathbb{T} \rangle} \; (\mathsf{T}{-}fix)$$

4 Abstract Executions

Here we present abstract executions, which we will use to record the run-time behaviour of programs.

We start by defining the behaviour of transactions at run-time. We assume a set of (run-time) transactions identifiers $\textsc{TransID} = \{\alpha, S, \cdots\}$ and a set of operations which we leave unspecified, though we require that $\mathsf{Op} \supseteq \left\{ \mathtt{read}\ a : v, \mathtt{write}\ a : v \;\middle|\; a \in \textsc{Addr} \land v \in \textsc{Val} \right\}$. We also assume a function $\mathsf{behav} : \textsc{TransID} \to \mathcal{P}(\mathsf{Op})$, which maps transactions into the operations that they perform on locations. With an abuse of notation, for any transaction α and operation o, we write $o \in \alpha$ (or $\alpha \ni o$) as a shorthand for $o \in \mathsf{behav}(\alpha)$. We only model transaction that enjoy *atomic visibility*. This means that **(i)** transactions never observe two different values when reading from the same location: $\forall \alpha \in$

$\textsc{TransID}, a \in \textsc{Addr}, v, v' \in \textsc{Val}.\, \alpha \ni \mathtt{read}\ a : v \wedge \alpha \ni \mathtt{read}\ a : v' \implies v = v'$; and **(ii)** the effects of a transactions become visible at once, which means that we never observe two different values written for the same location by a transaction: $\forall \alpha \in \textsc{TransID}, a \in \textsc{Addr}, v, v' \in \textsc{Val}.\, \alpha \ni \mathtt{write}\ a : v \wedge \alpha \ni \mathtt{write}\ a : v' \implies v = v'$.

Definition 4.1 (abstraction executions)**.** An abstract execution is a tuple $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{AR})$, where

- $\mathcal{T}$ is a finite, empty set of transactions,
- $\mathsf{SO} \subseteq \mathcal{T} \times \mathcal{T}$, the *program order*, is the union of disjoint, strict total orders over $\mathcal{T}$. That is, there exists a partition $\{\mathcal{T}_i\}_{i \in I}$ of $\mathcal{T}$ such that $\mathsf{SO} = \bigcup_{i \in I} \mathsf{SO}_i$, where for any $i \in I$, SO_i is a strict, total order over $\mathcal{T}_i$[1],
- $\mathsf{VIS} \subseteq \mathcal{T} \times \mathcal{T}$ is a strict, partial order such that $\mathsf{SO} \subseteq \mathsf{VIS}$, and $\mathsf{VIS} \,;\, \mathsf{VIS} \subseteq \mathsf{VIS}$,
- $\mathsf{AR} \subseteq \mathcal{T} \times \mathcal{T}$ is a strict, total order such that $\mathsf{VIS} \subseteq \mathsf{AR}$,
- for any location $a \in \textsc{Addr}$ let $\mathsf{Writes}(a) = \left\{ S \in \mathcal{T} \;\middle|\; S \ni \mathtt{write}\ a : \_ \right\}$. Given $T \in \mathcal{T}$, let also $\mathsf{previousWrites}_{\mathcal{X}}(a, \alpha) = \left\{ \alpha' \;\middle|\; \alpha' \in \mathsf{VIS}^{-1}(\alpha) \right\} \cap \mathsf{Writes}(a)$. Whenever $T \ni \mathtt{read}\ a : v$ for some transaction $\alpha \in \mathcal{T}$, location $a \in \textsc{Addr}$ and value $v \in \textsc{Val}$, then either $\mathsf{previousWrites}_{\mathcal{X}}(a, \alpha) = \emptyset$ and $v = 0$, or $\max_{\mathsf{AR}}(\mathsf{previousWrites}_{\mathcal{X}}(a, \alpha)) \ni \mathtt{write}\ a : v$.

The set of all abstract executions is denoted as $\mathsf{Executions}$. In the following, for an abstract execution $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{AR})$, we let $\mathcal{T}_{\mathcal{X}} = \mathcal{T}$, $\mathsf{SO}_{\mathcal{X}} = \mathsf{SO}$, $\mathsf{VIS}_{\mathcal{X}} = \mathsf{VIS}$, $\mathsf{AR}_{\mathcal{X}} = \mathsf{AR}$. We often use the notation $T \xrightarrow{R} S$ instead of $(T, S) \in R$.

Specification of Weak Consistency Models. We use the style of specification for weak consistency models proposed in [2].

Definition 4.2. A specification function ρ is an endo-function of relations over transactions, $\rho : (\mathbb{T} \times \mathbb{T}) \to (\mathbb{T} \times \mathbb{T})$, such that for any abstract execution $\mathcal{X}$ and relation $R \subseteq \mathcal{T}_{\mathcal{X}} \times \mathcal{T}_{\mathcal{X}}$, $\rho(R) = \rho(\mathcal{T}_{\mathcal{X}} \times \mathcal{T}_{\mathcal{X}}) \cap R?$.

A consistency guarantee is a pair (ρ, π) of specification functios. An abstract execution based specification of weak consistency models, or simply *x-specification*, is a (possibly empty, possibly infinite) set of consistency guarantees: $\mathsf{WCM} = \{(\rho_i, \pi_i)\}_{i \in I}$ for some index set I.

Definition 4.3. An abstract execution $\mathcal{X}$ is allowed by the consistency model specification WCM, written $\mathsf{WCM} \models \mathcal{X}$, if and only if, for any $(\rho, \pi) \in \mathsf{WCM}$, we have that $\rho(\mathsf{VIS}_{\mathcal{X}}) \,;\, \mathsf{AR}_{\mathcal{X}} \,;\, \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{AR}_{\mathcal{X}}$.

Example 4.4. Let $\rho_{\mathsf{Id}} = \lambda\_.\mathsf{Id}$, $\rho_{\mathsf{SI}} = \lambda R.(R \setminus \mathsf{Id})$, and $\rho_{[n]} = \lambda\_.[\mathsf{Writes}_{[n]}]$ for any location $[n] \in \mathit{Locs}$. Here, given a set $X \subseteq \mathbb{T}$, $[X]$ is defined as $\mathsf{Id} \cap (X \times X)$. We specify *Snapshot Isolation* via the set of consistency guarantees $\mathsf{WCM}_{\mathsf{SI}} = \{(\rho_{\mathsf{Id}}, \rho_{\mathsf{SI}})\} \cup \bigcup_{[n] \in \mathit{Locs}} \{(\rho_{[n]}, \rho_{[n]}\}$.

An abstract execution $\mathcal{X}$ is allowed by $\mathsf{WCM}_{\mathsf{SI}}$ if and only if $\mathsf{AR}_{\mathcal{X}} \,;\, \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, and for any $[n] \in \mathit{Locs}$, $[\mathsf{Writes}_{[n]}] \,;\, \mathsf{AR}_{\mathcal{X}} \,;\, [\mathsf{Writes}_{[n]}] \subseteq \mathsf{VIS}$.

SX: NEED TO REVISIT ABOVE LATER

[1] Recall that a relation $R \subseteq \mathcal{T} \times \mathcal{T}$ is a strict partial order if it is irreflexive and transitive. It is a strict total order if it enjoys the additional property that for any $T_1, T_2 \in \mathcal{T}$, either $T_1 = T_2$, $(T_1, T_2) \in R$ or $(T_2, T_1) \in R$.

In the following, we will use an incremental approach to build abstract executions from transactions. Suppose that an abstract execution $\mathcal{X}$ has been obtained as the (partial) result of a program P running in a system that implements the x-specification WCM. Suppose also that $T \in \mathcal{T}_{\mathcal{X}}$ is the transaction-instance associated with the last transactional code that has been executed by some thread, and that the same thread executes another piece of transactional code next, which results in the transaction instance S. This may result in an abstract execution $\mathcal{X}'$, where the new transaction instance S follows T in the program order $\mathsf{SO}_{\mathcal{X}'}$, and $\mathsf{VIS}_{\mathcal{X}'}, \mathsf{AR}_{\mathcal{X}'}$ are computed according to the axioms of WCM. $T \in \mathcal{T}_{\mathcal{X}}$. Note that the result of this procedure may result in an abstract execution that is not allowed by WCM, hence it is not always defined. Formally, we define an operator $+_{\mathsf{WCM}} : (\mathsf{Executions} \times \mathbb{T}) \rightharpoonup \mathsf{Executions}$ as follows:

Definition 4.5 (Runtime abstract executions)**.** Assuming set of thread identifiers $\textsc{ThreadID} \triangleq \{i, \ldots\}$, the set of *runtime abstract executions* is defined as the follows,

$$\hat{\mathcal{X}} \in \left\{ (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{AR}) \;\middle|\; \begin{array}{l} \mathcal{T} \in (\textsc{TransID} \uplus \textsc{ThreadID}) \rightharpoonup \mathcal{P}\left(\textsc{Events}\right) \\ \wedge\, \mathsf{SO}, \mathsf{VIS} \subseteq (\mathrm{dom}(\mathcal{T}) \cap \textsc{TransID}) \times \mathrm{dom}(\mathcal{T}) \\ \wedge\, \mathsf{AR} \subseteq (\mathrm{dom}(\mathcal{T}) \cap \textsc{TransID}) \times (\mathrm{dom}(\mathcal{T}) \cap \textsc{TransID}) \end{array} \right\}$$

Definition 4.6. Let WCM be a x-specification. Let $\mathcal{X} \in \mathsf{Executions}$, and let $T \in \mathcal{T}_{\mathcal{X}}$. Also, let $S \in \mathbb{T} \setminus \mathcal{T}_{\mathcal{X}}$. Define the abstract execution $\mathcal{X}'$ as follows: in which case we have

$$\begin{aligned} \mathsf{SO}'_{\mathcal{X}} &= (\mathsf{SO}_{\mathcal{X}} \cup \{(T, S)\})^{+} \\ \mathsf{AR}'_{\mathcal{X}} &= \{(T', S), (S, T'') \mid T' \in \mathsf{AR}^{-1}_{\mathcal{X}}(T) \wedge T'' \in \mathsf{AR}_{\mathcal{X}}(T)\}^{+} \\ \mathsf{VIS}'_{\mathcal{X}} &= \mu V. (\mathsf{SO}'_{\mathcal{X}} \cup \mathsf{VIS}_{\mathcal{X}} \cup \bigcup_{(\rho, \pi \in \mathsf{WCM})} \rho(V) \,;\, \mathsf{AR}'_{\mathcal{X}} \,;\, \pi(V))^{+} \end{aligned}$$

The abstract execution $(\mathcal{X}, T) +_{\mathsf{WCM}} S$ is defined to be exactly $\mathcal{X}'$ if whenever $S \ni \mathtt{write}\ [n] : \_$ and $S \xrightarrow{\mathsf{VIS}'_{\mathcal{X}}} T$, then $T \not\ni \mathtt{read}\ [n] : \_$ and $T \not\ni \mathtt{write}\ [n] : \_$, it is undefined otherwise.

Proposition 4.7. Let WCM be a x-specification, and suppose that $\mathsf{WCM} \models \mathcal{X}$ for some abstract execution $\mathcal{X}$. Let T, S be two transactions such that $(\mathcal{X}, T) +_{\mathsf{WCM}} S$ is defined. Then $\mathsf{WCM} \models (\mathcal{X}, T) +_{\mathsf{WCM}} S$.

ANDREA: Not sure whether it's true. Needs to be checked.

Note that, for a transaction T_0 such that $T_0 \ni o$ for no operation $o \in \mathsf{Op}$, $(\mathcal{X}, T) +_{\mathsf{WCM}} T_0$ is always defined (provided $T_0 \notin \mathcal{T}_{\mathcal{X}}, T \in \mathcal{T}_{\mathcal{X}}$).

Given an abstract execution and a x-specification $\mathcal{X}$, we can map any transaction $T \in \mathcal{T}_{\mathcal{X}}$ to a heap $h^{\mathsf{WCM}}_{\mathcal{X}}(T)$. Intuitively, the latter corresponds to the heap that would be observed by a client that interact with a system implementing the x-specification WCM, assuming that the set of transactions processed by the system so far resulted has given rise to the abstract execution $\mathcal{X}$, and that the last transaction instance of $\mathcal{X}$ observed by the client is T.

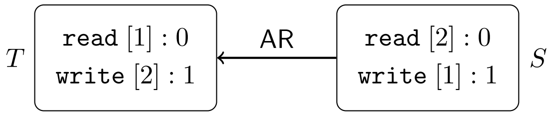
Definition 4.8. LetWCM be a consistency model specification, and $\mathcal{X}$ be an abstract execution. For any $T \in \mathcal{T}_{\mathcal{X}}$, let $\mathcal{X}' = (\mathcal{X}, T) +_{\mathsf{WCM}} T_0$, where $T_0 \notin \mathcal{T}_{\mathcal{X}}$ and $T_0 \ni o$ for no $o \in \mathsf{Op}$. We can always assume that such a transaction exists.

We define $h_{\mathcal{X}}^{\mathsf{WCM}}(T)$ as follows:

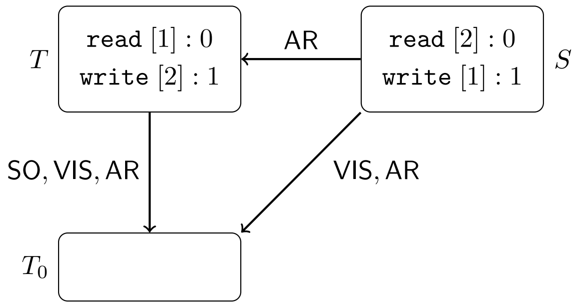
$$h_{\mathcal{X}}(T) = \lambda[n] \in \mathit{Locs}. \begin{cases} 0 & \Longleftarrow \ \mathsf{previousWrites}_{\mathcal{X}'}([n], T) = \emptyset \\ m & \Longleftarrow \ \max_{\mathsf{AR}_{\mathcal{X}'}}(\mathsf{previousWrites}_{\mathcal{X}'}([n], T_0)) \ni \mathtt{write}\ [n] : m \end{cases}$$

ANDREA: I don't really like this definition, but for the moment it will do...

Example 4.9. Consider the abstract execution $\mathcal{X}$ depicted below, denoting the write-skew anomaly allowed by snapshot isolation.

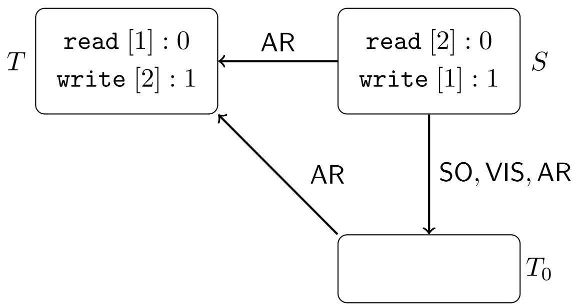


Let us calculate $h_{\mathcal{X}}^{\mathsf{WCM_{SI}}}(T)$. Let then T_0 be a transaction with no operation associated, and let us compute $\mathcal{X}' = (\mathcal{X}, T) +_{\mathsf{WCM_{SI}}} T_0$. By definition, we have that $T \xrightarrow{\mathsf{SO}_{\mathcal{X}'}} T_0$, hence $T \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} T_0$. Because $S \xrightarrow{\mathsf{AR}_{\mathcal{X}}} T$, we also have that $S \xrightarrow{\mathsf{AR}_{\mathcal{X}}} T$, and now from $S \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} T \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} T_0$, we obtain that $S \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} T_0$. By definition, we also know that $S \xrightarrow{\mathsf{AR}_{\mathcal{X}}} T$ implies that $S \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} T_0$. The final result is the abstract execution $\mathcal{X}'$ depicted below:



It is easy to see that $\mathsf{previousWrites}_{\mathcal{X}'}([1], T_0) = \{S\}$, and $\mathsf{previousWrites}_{\mathcal{X}'}([2], T_0) = \{T\}$. Because $S \ni \mathtt{write}\ [1] : 1$, we have that $h_{\mathcal{X}}^{\mathsf{WCM_{SI}}}(T)([1]) = 1$, and because $T \ni \mathtt{write}\ [2] : 1$, we have that $h_{\mathcal{X}}^{\mathsf{WCM_{SI}}}(T)([2]) = 1$.

Next, we want to calculate $h_{\mathcal{X}}^{\mathsf{WCM_{SI}}}(S)$. To this end, we need first to retrieve the abstract execution $\mathcal{X}'' = (\mathcal{X}, S) +_{\mathsf{WCM_{SI}}} T_0$, which is depicted below:



In this case we have that $\mathsf{previousWrites}_{\mathcal{X}''}([1], T_0) = \{S\}$, and $\mathsf{previousWrites}_{\mathcal{X}''}([2], T_0) = \emptyset$. By definition, we have that $h_{\mathcal{X}}^{\mathsf{WCM_{SI}}}(S)([1]) = 1$, and $h_{\mathcal{X}}^{\mathsf{WCM_{SI}}}(S)([2]) = 0$.

Semantics of Commands. Judgements for programs take the form $\langle \mathcal{X}, T, \sigma \mathsf{C}\rangle \rightarrow \langle \mathcal{X}', T', \sigma', \mathsf{C}'\rangle$.Here $\mathcal{X}$ is an abstract execution that represent the global run of the database, T represents the last transaction executed by the command, in the abstract execution, and σ is the thread local stack associated with the transaction.

The rule for evaluating a non-transactional primitive command is straightforward, as it does only require to manipulate the thread-local stack associated with the command.

$$\frac{\sigma \rightsquigarrow_c \sigma'}{\langle \mathcal{X}, T, \sigma, c.\mathsf{C}\rangle \rightarrow \langle \mathcal{X}, T, \sigma', \mathsf{C}\rangle}\ (prim{-}c)$$

Next, we give the rule for evaluating a transaction in a command of the form $[\mathbb{T}].\mathsf{C}$. First, given a read-set RS and a write-set WS, we define the set of transaction $\mathsf{makeTx}(\mathsf{RS}, \mathsf{WS})$ to be the largest set of transactions such that, whenever $T \in \mathsf{makeTx}(\mathsf{RS}, \mathsf{WS})$, then $T \ni \mathtt{read}\ [n] : m$, if and only if $([n], m) \in \mathsf{RS}$, and $T \ni \mathtt{write}\ [n] : m$ if and only if $([n], m) \in \mathsf{WS}$.

$$\frac{\begin{array}{c} T \xrightarrow{\mathsf{AR}_{\mathcal{X}}} S \quad T' \in (\mathsf{makeTx}(\mathsf{RS}, \mathsf{WS}) \setminus \mathcal{T}_{\mathcal{X}}) \quad \mathcal{X}' = (\mathcal{X}, S) +_{\mathsf{WCM}} T' \\ \sigma \vdash \langle \tau_0, h^{\mathsf{WCM}}_{\mathcal{X}}(S), \emptyset, \emptyset, \mathbb{T}\rangle \rightarrow^* \langle \tau', h', \mathsf{RS}, \mathsf{WS}, \mathbf{0}\rangle \end{array}}{\langle \mathcal{X}, T, \sigma, [\mathbb{T}].\mathsf{C}\rangle \rightarrow \langle \mathcal{X}', T', \sigma[\mathtt{ret} \mapsto \tau'(\mathtt{ret})], \mathsf{C}\rangle}\ (Tx{-}exec)$$

The three remaining rules are standard.

$$\frac{}{\langle \mathcal{X}, T, \sigma, \mathsf{C}_1 + \mathsf{C}_2\rangle \rightarrow \langle \mathcal{X}, \mathsf{C}_1, \sigma, \mathsf{C}\rangle}\ (C{-}choice{-}L)$$

$$\frac{}{\langle \mathcal{X}, T, \sigma, \mathsf{C}_1 + \mathsf{C}_2\rangle \rightarrow \langle \mathcal{X}, \mathsf{C}_2, \sigma, \mathsf{C}\rangle}\ (C{-}choice{-}R)$$

$$\frac{}{\langle \mathcal{X}, T, \sigma, \mu X.\mathsf{C}\rangle \rightarrow \langle \mathcal{X}, \{{}^{\mu X.\mathsf{C}}/_{X}\}\mathsf{C}, \sigma, \mathsf{C}\rangle}\ (C{-}fix)$$

Semantics of Programs.

References

[1] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory (CONCUR)*, pages 58–71. Dagstuhl, 2015.

[2] A. Cerone, A. Gotsman, and H. Yang. Algebraic laws for weak consistency. In *27th International Conference on Concurrency Theory (CONCUR)*, pages 26:1–22:16, 2017.