Sequential Composition for Relaxed Memory

Alan Jeffrey* and James Riely†
*The Servo Project and Roblox
†DePaul University

1. Model

1.1. Preliminaries

The syntax is built from

- a set of *values* $\mathcal{V}$, ranged over by v, w, ℓ,
- a set of *registers* $\mathcal{R}$, ranged over by r, s,
- a set of *expressions* $\mathcal{M}$, ranged over by M, N, L.

Memory locations are tagged values, written $[\ell]$. Let $\mathcal{X}$ be the set of memory locations, ranged over by x, y, z.

We require that

- values and registers are disjoint,
- values include at least the constants 0 and 1,
- expressions include at least registers and values,
- expressions do *not* include memory locations.

We model the following language.

$$\begin{array}{rl} \mu & ::= \mathsf{rlx} \mid \mathsf{ra} \mid \mathsf{sc} \\ C,\, D & ::= \mathtt{skip} \mid r{:=}M \mid r{:=}[L]^{\mu} \mid [L]^{\mu}{:=}M \\ & \quad \mid \mathtt{fork}\, G \mid C; D \mid \mathtt{if}(M)\{C\}\,\mathtt{else}\,\{D\} \\ G,\, H & ::= \mathsf{0} \mid \mathtt{thread}\, C \mid G \parallel H \end{array}$$

Memory modes, μ, are relaxed (rlx), release-acquire (ra), and sequentially consistent (sc). Relaxed is the default. *Commands*, C, include reads from and writes to memory at a given mode, as well as the usual structural constructs. *Thread groups*, G, include commands and 0, which denotes inaction. The `fork` command spawns a thread group. We often drop the words `fork` and `thread`.

The semantics is built from the following.

- a set of *actions* $\mathcal{A}$, ranged over by a,
- a set of *logical formulae* Φ, ranged over by ϕ, ψ.

We require that

- actions include writes $(\mathsf{W}xv)$ and reads $(\mathsf{R}xv)$,
- formulae include equalities $(M{=}v)$ and $(M{=}x)$,
- formulae are closed under negation, conjunction, disjunction, and substitutions $[x/r]$ and $[M/x]$,
- there is an entailment relation $\vDash$ between formulae, with the expected semantics.

Logical formulae include equations over locations and registers, such $(x{=}1)$ and $(r{=}s{+}1)$. We use expressions as formulae, coercing M to $M \neq 0$.

Formulae are *open*, in that occurrences of register names and memory locations are subject to substitutions of the form $\phi[x/r]$ and $\phi[N/x]$. Actions are not subject to substitution.

We say ϕ *implies* ψ if $\phi \vDash \psi$. We say ϕ is a *tautology* if $\mathsf{true} \vDash \phi$. We say ϕ is *unsatisfiable* if $\phi \vDash \mathsf{false}$.

1.2. Pomsets

We first consider a fragment of our language that can be modeled using simple pomsets.

Definition 1. A *pomset* over $\mathcal{A}$ is a tuple $(E, \leq, \lambda)$ where

- E is a set of *events*,
- $\leq \subseteq (E \times E)$ is the *causality* partial order,
- $\lambda : E \to \mathcal{A}$ is a *labeling*.

Let P range over pomsets, and $\mathcal{P}$ over sets of pomsets.

We lift terminology from actions to events. For example, we say that e writes x if $\lambda(e)$ writes x. We also drop quantifiers when clear from context, such as $(\forall e \in E)(\forall x \in \mathcal{X})$.

Definition 2. Action $(\mathsf{W}xv)$ *matches* $(\mathsf{R}xw)$ when $v = w$. Action $(\mathsf{W}xv)$ *blocks* $(\mathsf{R}xw)$ when $v \neq w$.

We say that e is *fulfilled* if there is a $d \leq e$ which matches it and, for any c which can block e, either $c \leq d$ or $e \leq c$.

We say that P is *fulfilled* if every read in P is fulfilled.

We define *independency* $(\leftrightarrow \subseteq \mathcal{A} \times \mathcal{A})$ as follows.

$$\begin{array}{l} \leftrightarrow = \{(\mathsf{R}xv, \mathsf{R}yw)\} \\ \quad \cup \{(\mathsf{W}xv, \mathsf{W}yw) \mid x \neq y \vee v = w\} \\ \quad \cup \{(\mathsf{R}xv, \mathsf{W}yw), (\mathsf{W}xv, \mathsf{R}yw) \mid x \neq y\} \end{array}$$

In order to give the semantics, we define several operators over sets of pomsets.

Definition 3.
If $P \in \mathit{STOP}$ then $E = \emptyset$.
If $P \in (\mathcal{P}_1 \parallel \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

1) $E = (E_1 \cup E_2)$,
2) if $e \in E_1$ then $\lambda(e) = \lambda_1(e)$,
3) if $e \in E_2$ then $\lambda(e) = \lambda_2(e)$,
4) if $d \leq_1 e$ then $d \leq e$,
5) if $d \leq_2 e$ then $d \leq e$,
6) E_1 and E_2 are disjoint.

If $P \in (a \to \mathcal{P})$ then $(\exists P_2 \in \mathcal{P})$

1) $E = (E_1 \cup E_2)$,

2) if $e \in E_1$ then $\lambda(e) = a$,
3) if $e \in E_2$ then $\lambda(e) = \lambda_2(e)$,
4) if $d, e \in E_1$ then $d = e$,
5) if $d \leq_2 e$ then $d \leq e$,
6) if $d \in E_1$ and $e \in E_2$, either $d < e$ or $a \leftrightarrow \lambda_2(e)$.

Using these operators, we can give the semantics for a simple fragment of our language.

$$\begin{aligned}
[\![0]\!] &= \mathit{STOP} \\
[\![G \parallel H]\!] &= [\![G]\!] \parallel [\![H]\!] \\
[\![x{:=}v; C]\!] &= (\mathsf{W}xv) \to [\![C]\!] \\
[\![r{:=}x; C]\!] &= \textstyle\bigcup_v (\mathsf{R}xv) \to [\![C]\!]
\end{aligned}$$

If we take $\leftrightarrow = \emptyset$, then we have sequentially consistent execution.

[Do Examples.]

[Do examples with coherence.]

[Note that this allows mumbling for reads and writes.]

[Use refinement (that is subset order) as notion of compiler optimization.]

[Talk about Mazurkiewicz traces.]

1.3. Pomsets with Preconditions

[Problem with previous section is that notion of dependency is impoverished]

Definition 4. A *pomset with preconditions* is a pomset together with $\kappa : E \to \Phi$.

Definition 5. A pomset with preconditions is *top level* if it is fulfilled and every precondition is a tautology.

Definition 6.
If $P \in \mathit{STOP}$ then $E = \emptyset$.
If $P \in (\mathcal{P}_1 \parallel \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

1–6) as for $\parallel$ in Definition 3,
7) if $e \in E_1$ then $\kappa(e)$ implies $\kappa_1(e)$,
8) if $e \in E_2$ then $\kappa(e)$ implies $\kappa_2(e)$.

If $P \in \mathit{IF}(\psi, \mathcal{P}_1, \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

1–5) as for $\parallel$ in Definition 3 (ignoring disjointness),
6) if $e \in E_1 \setminus E_2$ then $\kappa(e)$ implies $\psi \wedge \kappa_1(e)$,
7) if $e \in E_2 \setminus E_1$ then $\kappa(e)$ implies $\neg\psi \wedge \kappa_2(e)$,
8) if $e \in E_1 \cap E_2$ then
$\kappa(e)$ implies $(\psi \wedge \kappa_1(e)) \vee (\neg\psi \wedge \kappa_2(e))$.

If $P \in \mathit{STOREPRE}(x, M, \mathcal{P}_2)$ then $(\exists P_2 \in \mathcal{P}_2)\ (\exists v \in \mathcal{V})$

1–6) as for $(\mathsf{W}xv) \to P_2$ in Definition 3,
7) if $d \in E_1 \setminus E_2$ then $\kappa(d)$ implies $(M{=}v)$,
8) if $d \in E_1$ and $e \in E_2$ then either
$\kappa(e)$ implies $(M{=}v) \vee \kappa_2(e)$ and $d = e$ or
$\kappa(e)$ implies $\kappa_2(e)$.

If $P \in \mathit{LOADPRE}(x, r, \mathcal{P}_2)$ then $(\exists P_2 \in \mathcal{P}_2)\ (\exists v \in \mathcal{V})$

1–6) as for $(\mathsf{R}xv) \to P_2$ in Definition 3,
7) if $d \in E_1$ and $e \in E_2$ then either $d = e$ or
$\kappa(e)$ implies $(r{=}v \vee r{=}x) \Rightarrow \kappa_2(e)[r/x]$ or
$\kappa(e)$ implies $(r{=}v) \Rightarrow \kappa_2(e)[r/x]$ and $d < e$.

Note that when $d \in E_1 \setminus E_2$, *LOADPRE* does not constrain $\kappa(d)$.

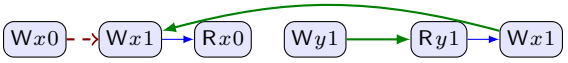
The semantics of 0 and $\parallel$ are as before.

$$\begin{aligned}
[\![\mathtt{if}(\psi)\{C\}\,\mathtt{else}\,\{D\}]\!] &= \mathit{IF}(\psi, [\![C]\!], [\![D]\!]) \\
[\![r{:=}M; C]\!] &= [\![C]\!][M/r] \\
[\![x{:=}M; C]\!] &= \mathit{STOREPRE}(x, M, [\![C]\!]) \\
[\![r{:=}x; C]\!] &= \mathit{LOADPRE}(x, r, [\![r]\!])
\end{aligned}$$

This is essentially the model of Jagadeesan et al. [2020], restricted to relaxed access. There are only two substantial differences. First, for simplicity, we enforce read-read dependencies. Second, in the rule for read prefixing we have substituted $[r/x]$, rather than $[x/r]$. This means that reads clobber local state. We assume registers are only used once—otherwise, one needs to generate a fresh register for the substitution.

With read-read dependencies, the second difference can be seen. For example, the following execution is allowed with $[x/r]$, but not $[r/x]$.

$$x{:=}0; r{:=}x; \mathtt{if}(r)\{s{:=}x\}; y{:=}s{+}1 \parallel x{:=}y$$



[Is there a difference w/o read-read dependencies?]

[Stuff about conditionals and merging events.]

1.4. Pomsets with Predicate Transformers

[The problem with the previous section is that there's no story for sequential composition.]

Definition 7. A *predicate transformer* is a monotone function $\tau : \Phi \to \Phi$ such that $\tau(\mathsf{false})$ is false, $\tau(\phi \wedge \psi)$ is $\tau(\phi) \wedge \tau(\psi)$, and $\tau(\phi \vee \psi)$ is $\tau(\phi) \vee \tau(\psi)$.

Definition 8. A *family of predicate transformers* indexed by subsets of E consists of predicate transformers τ^D for each set of events D, such that if $C \subseteq D$ then $\tau^C(\phi)$ implies $\tau^D(\phi)$.

Definition 9. A pomset with predicate tansformers is a pomset with preconditions, together with a family of predicate transformers τ indexed by subsets of E.

Define *THREAD* to embed pomsets with predicate transformers into pomsets with preconditions simply by dropping the predicate transformer. For the reverse embedding, *FORK* adopts the identity transformer.

Definition 10. If $P \in \mathit{FORK}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$

1) $E = E_1$,
2) $\lambda(e) = \lambda_1(e)$,
3) $\kappa(e)$ implies $\kappa_1(e)$,
4) $\tau^D(\phi)$ implies ϕ.

Definition 11. If $P \in \mathit{STOP}$ then $E = \emptyset$ and

1) $\tau^D(\phi)$ implies false.

If $P \in \mathit{SKIP}$ then $E = \emptyset$ and

1) $\tau^D(\phi)$ implies ϕ.

If $P \in \mathit{LET}(r,\ M)$ then $E = \emptyset$ and

1) $\tau^D(\phi)$ implies $\phi[M/r]$.

If $P \in \mathit{IF}(\psi,\ \mathcal{P}_1,\ \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

1–8) as for *IF* in Definition 6,
9) $\tau^D(\phi)$ implies $(\psi \wedge \tau_1^D(e)) \vee (\neg\psi \wedge \tau_2^D(\phi))$.

If $P \in (\mathcal{P}_1\,;\mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$,

1–5) as for $\|$ in Definition 3 (ignoring disjointness),
6) if $e \in E_1 \setminus E_2$ then $\kappa(e)$ implies $\kappa_1(e)$,
7) if $e \in E_2 \setminus E_1$ then $\kappa(e)$ implies $\kappa_2'(e)$,
8) if $e \in E_1 \cap E_2$ then $\kappa(e)$ implies $\kappa_1(e) \vee \kappa_2'(e)$, where $\kappa_2'(e) = \tau_1^C(\kappa_2(e))$, where $C = \{c \mid c < e\}$,
9) $\tau^D(\phi)$ implies $\tau_2^D(\tau_1^D(\phi))$,
10) if $d \in E_1$ and $e \in E_2$ either $d < e$ or $a \leftrightarrow \lambda_2(e)$.

If $P \in \mathit{STORE}(x,\ M)$ then $(\exists v \in \mathcal{V})$

1) $\lambda(e) = (\mathsf{W}xv)$,
2) $\kappa(e)$ implies $(M{=}v)$,
3) $\tau^{\emptyset}(\phi)$ implies $\phi[M/x]$,
4) $\tau^D(\phi)$ implies $(M{=}v) \wedge \phi[M/x]$, if $D \neq \emptyset$,
5) if $d, e \in E$ then $d = e$.

If $P \in \mathit{LOAD}(x,\ r)$ then $(\exists v \in \mathcal{V})$

1) $\lambda(e) = (\mathsf{R}xv)$,
2) $\tau^{\emptyset}(\phi)$ implies $(r{=}v \vee r{=}x) \Rightarrow \phi[r/x]$,
3) $\tau^D(\phi)$ implies $(r{=}v) \Rightarrow \phi[r/x]$, if $D \neq \emptyset$,
4) if $d, e \in E$ then $d = e$.

The complete semantics is as follows.

$$
\begin{aligned}
[\![\mathtt{0}]\!] &= \mathit{STOP} \\
[\![\mathtt{skip}]\!] &= \mathit{SKIP} \\
[\![r{:=}x]\!] &= \mathit{LOAD}(x,\ r) \\
[\![x{:=}M]\!] &= \mathit{STORE}(x,\ M) \\
[\![r{:=}M]\!] &= [\![C]\!][M/r] \\
[\![C;D]\!] &= [\![C]\!]\,;[\![D]\!] \\
[\![G \parallel H]\!] &= [\![G]\!] \,\|\, [\![H]\!] \\
[\![\mathtt{if}(\psi)\{C\}\,\mathtt{else}\,\{D\}]\!] &= \mathit{IF}(\psi,\ [\![C]\!],\ [\![D]\!]) \\
[\![\mathtt{fork}\ G]\!] &= \mathit{FORK}[\![G]\!] \\
[\![\mathtt{thread}\ C]\!] &= \mathit{THREAD}[\![C]\!]
\end{aligned}
$$

[Examples.]

[Skolemization ensures disjunction closure, which is necessary for associativity. Show example.]

2. Complications

[Very drafty. These are just notes.]
Complications:

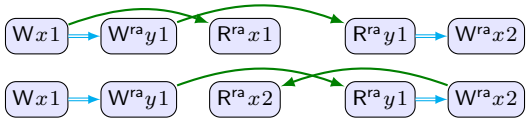
- Dependency: preconditions
- TC1: Track local state
- Release/Acquire: Q
- Coherence respects program order: Q_x
- Drop read-read coherence: QW_x (Required for CSE without alias analysis over read only code, not required by hardware)
- Address calculation...
- No read-read dependency (for arm): RW/RO (control dependencies into reads as in MP with release on right and control dependency on left)
- Downgrading acquires/Anton example: $\downarrow x$

Require indexing by event:

- IF closure/case analysis: ψ_e
- Redundant read elimination/TC2, to get determinism of value per event

2.1. Triangle Races

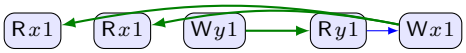
$$x{:=}1; y^{\mathsf{ra}}{:=}1; r{:=}x^{\mathsf{ra}} \parallel \mathtt{if}(y^{\mathsf{ra}})\{x^{\mathsf{ra}}{:=}2\}$$



Bug is in [Dongol et al., 2019, Lemma A.4]. It assumes that $(\mathsf{R}^{\mathsf{ra}}x1)$ and $(\mathsf{W}^{\mathsf{ra}}x2)$ are racing in the first execution because they are not ordered by happens-before. But this is false since neither is plain.

2.2. TC2

$$r{:=}x; s{:=}x; \mathtt{if}(r{=}s)\{y{:=}1\} \parallel x{:=}y \qquad \text{(TC2)}$$



Precondition of $(\mathsf{W}y1)$ is $(r{=}s)$ in $[\![\mathtt{if}(r{=}s)\{y{:=}1\}]\!]$. Predicate transformers for $\emptyset$ in $[\![r{:=}x]\!]$ and $[\![s{:=}x]\!]$ are

$$\langle (r{=}1 \vee r{=}x) \Rightarrow \phi[r/x] \mid \phi\rangle,$$
$$\langle (s{=}1 \vee s{=}x) \Rightarrow \phi[s/x] \mid \phi\rangle.$$

Combining the transformers, we have

$$\langle (r{=}1 \vee r{=}x) \Rightarrow (s{=}1 \vee s{=}r) \Rightarrow \phi[s/x] \mid \phi\rangle.$$

Applying this to $(r{=}s)$, we have

$$\langle (r{=}1 \vee r{=}x) \Rightarrow (s{=}1 \vee s{=}r) \Rightarrow (r{=}s) \mid \phi\rangle,$$

which is not a tautology.

Same problem occurs oopsla, where we have:

$$\langle \phi[v/x, r] \wedge \phi[x/r] \mid \phi\rangle,$$
$$\langle \phi[v/x, s] \wedge \phi[x/s] \mid \phi\rangle.$$

Combining the transformers, we have

$$\langle \phi[v/x, r, s] \wedge \phi[v/x, r][x/s] \wedge \phi[x/r][v/x, s] \wedge \phi[x/r, s] \mid \phi\rangle.$$

Applying this to $(r{=}s)$, we have

$$\langle v{=}v \wedge v{=}x \wedge x{=}v \wedge x{=}x \mid \phi\rangle,$$

which is not a tautology.

The semantics here allows this by coalescing:

$$r{:=}\,x;\, s{:=}\,x;\, \texttt{if}(r{=}s)\{y{:=}\,1\} \parallel x{:=}\,y$$

2.3. Coherence

Can be encoded in independency, or logic.

2.4. Release Acquire

Can be encoded in independency, or logic, but logic is more flexible, and we need that for ARM8.

2.5. Agda

References

B. Dongol, R. Jagadeesan, and J. Riely. Modular transactions: bounding mixed races in space and time. In J. K. Hollingsworth and I. Keidar, editors, *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 82–93. ACM, 2019. doi: 10.1145/3293883.3295708. URL https://doi.org/10.1145/3293883.3295708.

R. Jagadeesan, A. Jeffrey, and J. Riely. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.*, 4(OOPSLA):194:1–194:30, 2020. doi: 10.1145/3428262. URL https://doi.org/10.1145/3428262.

```
κLOAD : Address → Value → Formula
κLOAD a v = RO ∧ Qw[ a ]

τLOADD : Register → Register → Address → Value → Formula → Formula
τLOADD r s a v ϕ = (value v == register s) ⇒ (ϕ [ register s / r ] [ register s /[ a ]])

τLOADI : Register → Register → Address → AccessMode → Formula → Formula
τLOADI r s a rlx ϕ =          ¬ Q[ a ] ∧ (RW ⇒ ([ a ]== register s) ⇒ (ϕ [ register s / r ] [ register s /[ a ]]))
τLOADI r s a ra  ϕ = ↓[ a ] ∧ ¬ Q[ a ] ∧ (RW ⇒ ([ a ]== register s) ⇒ (ϕ [ register s / r ] [ register s /[ a ]]))

τLOADø : Register → Register → Address → AccessMode → Formula → Formula
τLOADø r s a rlx ϕ =          ¬ Q[ a ] ∧ (ϕ [ register s / r ] [ register s /[ a ]])
τLOADø r s a ra  ϕ = ↓[ a ] ∧ ¬ Q[ a ] ∧ (ϕ [ register s / r ] [ register s /[ a ]])

record LOAD (r : Register) (L : Expression) (μ : AccessMode) (P : PomsetWithPredicateTransformers) : Set₁ where

  open PomsetWithPredicateTransformers P

  field a : Event → Address
  field v : Event → Value
  field ψ : Event → Formula

  field d=e : ∀ d e → (d ∈ E) → (e ∈ E) → ((ψ(d) ∧ ψ(e)) ∈ Satisfiable) → (d ≡ e)
  field ℓ=Rav : ∀ e → (e ∈ E) → ℓ(e) ≡ (R (a(e)) (v(e)))
  field κ⊨κLOAD :  ∀ e → (e ∈ E) → κ(e) ⊨ (ψ(e) ∧ (L == address (a(e))) ∧ κLOAD (a(e)) (v(e)))
  field τC⊨τLOADD : ∀ C ϕ e → (e ∈ E) → (e ∈ C) → (τ(C)(ϕ) ⊨ (ψ(e) ⇒ τLOADD r (r[ e ]) (a(e)) (v(e)) ϕ))
  field τC⊨τLOADI : ∀ C ϕ a e → (e ∈ E) → (e ∉ C) → (τ(C)(ϕ) ⊨ (ψ(e) ⇒ (L == address a) ⇒ τLOADI r (r[ e ]) a μ ϕ))
  field τC⊨τLOADø : ∀ C ϕ a s χ → (∀ e → (e ∈ E) → (e ∈ C) → (χ ⊨ ¬(ψ(e)))) → (τ(C)(ϕ) ⊨ (χ ⇒ (L == address a) ⇒ τLOADø r s a μ ϕ))

κSTORE : AccessMode → Expression → Address → Value → Formula
κSTORE rlx M a v = (M == value v) ∧ RW ∧ Q[ a ]
κSTORE ra  M a v = (M == value v) ∧ RW ∧ Q

τSTORED : AccessMode → Expression → Address → Value → Formula → Formula
τSTORED rlx M a v ϕ = (Qw[ a ] ⇒ (M == value v)) ∧ (ϕ [ M /[ a ]] [ tt /↓[ a ]])
τSTORED ra  M a v ϕ = (Qw[ a ] ⇒ (M == value v)) ∧ (ϕ [ M /[ a ]] [ ff /↓[*]])

τSTOREI : AccessMode → Expression → Address → Formula → Formula
τSTOREI rlx M a ϕ = (¬ Qw[ a ]) ∧ (ϕ [ M /[ a ]] [ tt /↓[ a ]])
τSTOREI ra  M a ϕ = (¬ Qw[ a ]) ∧ (ϕ [ M /[ a ]] [ ff /↓[*]])

record STORE (L : Expression) (μ : AccessMode) (M : Expression) (P : PomsetWithPredicateTransformers) : Set₁ where

  open PomsetWithPredicateTransformers P

  field a : Event → Address
  field v : Event → Value
  field ψ : Event → Formula

  field d=e : ∀ d e → (d ∈ E) → (e ∈ E) → ((ψ(d) ∧ ψ(e)) ∈ Satisfiable) → (d ≡ e)
  field ℓ=Wav : ∀ e → (e ∈ E) → ℓ(e) ≡ (W (a(e)) (v(e)))
  field κ⊨κSTORE :  ∀ e → (e ∈ E) → (κ(e) ⊨ (ψ(e) ∧ (L == address (a(e))) ∧ κSTORE μ M (a(e)) (v(e))))
  field τC⊨τSTORED : ∀ C ϕ a e → (e ∈ E) → (e ∈ C) → (τ(C)(ϕ) ⊨ (ψ(e) ⇒ (L == address a) ⇒ τSTORED μ M a (v(e)) ϕ))
  field τC⊨τSTOREI : ∀ C ϕ a χ → (∀ e → (e ∈ E) → (e ∈ C) → (χ ⊨ ¬(ψ(e)))) → (τ(C)(ϕ) ⊨ (χ ⇒ (L == address a) ⇒ τSTOREI μ M a ϕ))
```