Program logics for relaxed consistency
UPMARC Summer School 2014

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

2nd Lecture, 29 July 2014

Recap

Topics covered yesterday:

- The C11 memory model
- Separation logic
- Relaxed separation logic

Today:

- Compare and swap
- GPS
- Advanced features

Recap: Rules for release/acquire accesses

Ownership transfer by rel-acq synchronizations.

- Atomic allocation $\leadsto$ pick loc. invariant $\mathcal{Q}$.

$$\{\mathcal{Q}(v)\}\ x = \mathsf{alloc}(v);\ \{\mathbf{W}_{\mathcal{Q}}(x) * \mathbf{R}_{\mathcal{Q}}(x)\}$$

- Release write $\leadsto$ give away permissions.

$$\{\mathbf{W}_{\mathcal{Q}}(x) * \mathcal{Q}(v)\}\ x.\mathsf{store}(v, \mathit{rel});\ \{\mathbf{W}_{\mathcal{Q}}(x)\}$$

- Acquire read $\leadsto$ gain permissions.

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ t = x.\mathsf{load}(\mathit{acq});\ \{\mathcal{Q}(t) * \mathbf{R}_{\mathcal{Q}[t:=\mathsf{emp}]}(x)\}$$

Basically, disallow ownership transfer.

- Relaxed reads:

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ t = x.load(rlx)\ \left\{\begin{array}{l}\mathbf{R}_{\mathcal{Q}}(x) \wedge \\ (\mathcal{Q}(t) \not\equiv \text{false})\end{array}\right\}$$

- Relaxed writes:

$$\frac{\mathcal{Q}(v) = \text{emp}}{\{\mathbf{W}_{\mathcal{Q}}(x)\}\ x.store(v, rlx)\ \{\mathbf{W}_{\mathcal{Q}}(x)\}}$$

Compare and swap (CAS)

A standard primitive for implementing concurrent algorithms

$$
\begin{array}{l}
x.\mathrm{CAS}(v, v', M) \stackrel{\text{def}}{=} \\
\qquad \textbf{atomic } \{ \\
\qquad\quad \textbf{if}(x.\mathrm{load}(M) == v)\{ \\
\qquad\qquad x.\mathrm{store}(v', M); \\
\qquad\qquad \textbf{return } \mathrm{true}; \\
\qquad\quad \} \\
\qquad\quad \textbf{return } \mathrm{false}; \\
\qquad \}
\end{array}
$$

Reasoning about CAS in RSL

- New assertion form, $P := \ldots \mid \mathbf{C}_{\mathcal{Q}}(x)$.
- "Permission to do a CAS"
- Duplicable:

$$\mathbf{C}_{\mathcal{Q}}(x) \iff \mathbf{C}_{\mathcal{Q}}(x) * \mathbf{C}_{\mathcal{Q}}(x)$$

- Also allows writing:

$$\mathbf{C}_{\mathcal{Q}}(x) \iff \mathbf{C}_{\mathcal{Q}}(x) * \mathbf{W}_{\mathcal{Q}}(x)$$

- And reading without ownership transfer:

$$\mathbf{C}_{\mathcal{Q}}(x) \iff \mathbf{C}_{\mathcal{Q}}(x) * \mathbf{R}_{\mathrm{emp}}(x)$$

Reasoning about CAS in RSL

Allocation rule:

$$\{\mathcal{Q}(v)\}\ x = \mathsf{alloc}(v);\ \{\mathbf{C}_{\mathcal{Q}}(x)\}$$

CAS rule:

$$\frac{\begin{array}{c} t \land P * \mathcal{Q}(v) \Rightarrow \mathcal{Q}(v') * R \\ \neg t \land P \Rightarrow R \\ X \in \{rel, rlx\} \Rightarrow \mathcal{Q}(v) \equiv \mathsf{emp} \\ X \in \{acq, rlx\} \Rightarrow \mathcal{Q}(v') \equiv \mathsf{emp} \end{array}}{\{\mathbf{C}_{\mathcal{Q}}(x) * P\}\ t = x.\mathsf{CAS}(v, v', X)\ \{R\}}$$

Mutual exclusion locks

Attach a 'resource invariant' at each lock:

$$Lock(x, J) \iff Lock(x, J) * Lock(x, J)$$

Specifications for mutex operations:

$$\{J\}\ x = \textit{new-lock}()\ \{Lock(x, J)\}$$

$$\{Lock(x, J)\} \quad lock(x) \quad \{Lock(x, J) * J\}$$

$$\{J * Lock(x, J)\} \quad unlock(x) \quad \{Lock(x, J)\}$$

Mutual exclusion locks

Let $\mathcal{Q}_J(v) \stackrel{\text{def}}{=} (v = 0 \land \mathsf{emp}) \lor (v = 1 \land J)$

$Lock(x, J) \stackrel{\text{def}}{=} \mathbf{C}_{\mathcal{Q}_J}(x)$

$new\text{-}lock() \stackrel{\text{def}}{=}$

$\{J\}$

$res = \mathbf{alloc}(1)$

$\{Lock(res, J)\}$

$unlock(x) \stackrel{\text{def}}{=}$

$\{J * Lock(x, J)\}$

$x.\mathsf{store}(1, rel)$

$\{Lock(x, J)\}$

$lock(x) \stackrel{\text{def}}{=}$

$\{Lock(x, J)\}$

repeat

$\{Lock(x, J)\}$

$t = x.\mathsf{CAS}(1, 0, acq)$

$\{Lock(x, J) * (t \Rightarrow J)\}$

until t

$\{J * Lock(x, J)\}$

GPS: Towards a better logic for C11

- Protocols
- Ghosts & escrows

GPS: A better logic for release-acquire

Three key features:

- Location ~~invariants~~ **protocols**
- Ghost state/tokens 
- Escrows for ownership transfer

Example (Racy message passing)

Initially, $x = y = 0$.

$$\begin{array}{l} x.\mathrm{store}(1, \mathit{rel}); \\ y.\mathrm{store}(1, \mathit{rel}); \end{array} \left\| \begin{array}{l} x.\mathrm{store}(1, \mathit{rel}); \\ y.\mathrm{store}(1, \mathit{rel}); \end{array} \right\| \begin{array}{l} t = y.\mathrm{load}(\mathit{acq}); \\ t' = x.\mathrm{load}(\mathit{acq}); \end{array}$$

Cannot get $t = 1 \wedge t' = 0$.

Racy message passing in GPS

Protocol for x: $\boxed{\textbf{A: } x = 0} \longrightarrow \boxed{\textbf{B: } x = 1}$

Protocol for y: $\boxed{\textbf{C: } y = 0} \longrightarrow \boxed{\textbf{D: } y = 1 \land x.st \geq \textbf{B}}$

Acquire reads gain knowledge, not ownership.

$$
\begin{array}{l}
\{x.st \geq \textbf{A} \land y.st \geq \textbf{C}\} \\
x.\text{store}(1, rel); \\
\{x.st \geq \textbf{B} \land y.st \geq \textbf{C}\} \\
y.\text{store}(1, rel); \\
\{x.st \geq \textbf{B} \land y.st \geq \textbf{D}\}
\end{array}
\;\Bigg\|\;
\begin{array}{l}
\{x.st \geq \textbf{A} \land y.st \geq \textbf{C}\} \\
t = y.\text{load}(acq); \\
\left\{\begin{array}{l} t = 0 \land x.st \geq \textbf{A} \\ \lor\, t = 1 \land x.st \geq \textbf{B} \end{array}\right\} \\
t' = x.\text{load}(acq); \\
\{t = 0 \lor (t = 1 \land t' = 1)\}
\end{array}
$$

Rules for reads and writes

Read rule:

$$\frac{\begin{array}{l}\forall s' \geq_\tau s.\ \ \mathbf{inv}_\tau(s', t) * P \Rightarrow Q \\ Q \Leftrightarrow Q * Q\end{array}}{\left\{\begin{array}{c}x.st \geq_\tau s \\ * P\end{array}\right\}\ t = x.\mathrm{load}(acq);\ \left\{\begin{array}{c}\exists s'.\ x.st \geq_\tau s' \\ * P * Q\end{array}\right\}}$$

Write rule:

$$\frac{\begin{array}{l}P \Rrightarrow \mathbf{inv}_\tau(s'', v) * Q \\ \forall s' \geq_\tau s.\ \ \mathbf{inv}_\tau(s', \_) * P \Rightarrow s'' \geq_\tau s'\end{array}}{\{x.st \geq_\tau s * P\}\ x.\mathrm{store}(v, rel);\ \{x.st \geq_\tau s'' * Q\}}$$

GPS ghosts and escrows

We can create ghost unduplicable tokens:

$$\frac{K \text{ is fresh}}{P \Rrightarrow P * K} \qquad \frac{}{K * K \Rightarrow \text{false}}$$

We can also create escrows:

$$\frac{P * P \Rightarrow \text{false}}{Q \Rrightarrow \textbf{Esc}(P, Q)} \qquad \frac{}{\textbf{Esc}(P, Q) * P \Rrightarrow Q}$$

Escrows are duplicable:

$$\textbf{Esc}(P, Q) \iff \textbf{Esc}(P, Q) * \textbf{Esc}(P, Q)$$

but only one component can 'unlock' them.

To gain ownership, we use ghost state & escrows.

$$\frac{P * P \Rightarrow \text{false}}{Q \Rrightarrow \mathbf{Esc}(P, Q)} \qquad \frac{}{\mathbf{Esc}(P, Q) * P \Rrightarrow Q}$$

Example (Message passing using escrows)

Invariant for x: $x = 0 \vee \mathbf{Esc}(K, \&a \mapsto 7)$.

$\{\&a \mapsto 0\}$	$\{K\}$
$a = 7;$	**if** $(x.\text{load}(acq) \neq 0)$
$\{\&a \mapsto 7\}$	$\quad\{K * \mathbf{Esc}(K, \&a \mapsto 7)\}$
$\{\mathbf{Esc}(K, \&a \mapsto 7)\}$	$\quad\{\&a \mapsto 7\}$
$x.\text{store}(1, rel);$	$\quad$ **print**$(a);$

Rule for CAS

With a successful CAS we can gain not only knowledge, but also ownership:

$$\frac{\begin{array}{l}\forall s'' \geq_\tau s.\ \ \mathbf{inv}_\tau(s'', v) * P \Rrightarrow \mathbf{inv}_\tau(s', v') * Q \wedge s' \geq_\tau s'' \\ \forall s'' \geq_\tau s.\ \forall v'' \neq v.\ \ \mathbf{inv}_\tau(s'', v'') * P \Rightarrow R \\ R \Leftrightarrow R * R\end{array}}{\left\{\begin{array}{c}x.st \geq_\tau s \\ * P\end{array}\right\}\ \begin{array}{l}t = x.\mathrm{CAS} \\ (v, v', \mathit{rel\text{-}acq});\end{array}\ \left\{\begin{array}{l}(t \wedge x.st \geq_\tau s' * Q) \\ \vee \neg t \wedge P * R\end{array}\right\}}$$

Reasoning about advanced C11 features

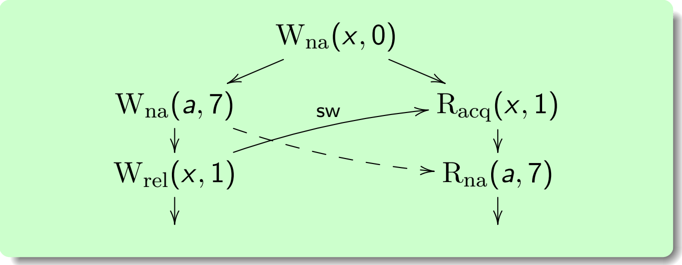
(Work in progress)

- Fences
- Consume reads

Message passing

$$\textbf{int } a;\ \textbf{atomic\_int } x = 0;$$

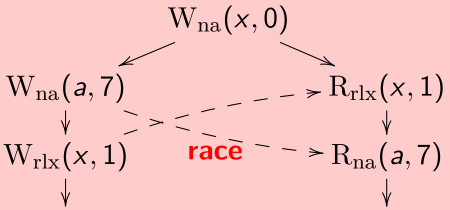
$$\left(\begin{array}{l\|l} a = 7; & \textbf{if } (x.\mathrm{load}(acq) \neq 0)\{ \\ x.\mathrm{store}(1, rel); & \quad \textbf{print}(a);\} \end{array}\right)$$



Incorrect message passing

$$\textbf{int } a;\ \textbf{atomic\_int } x = 0;$$

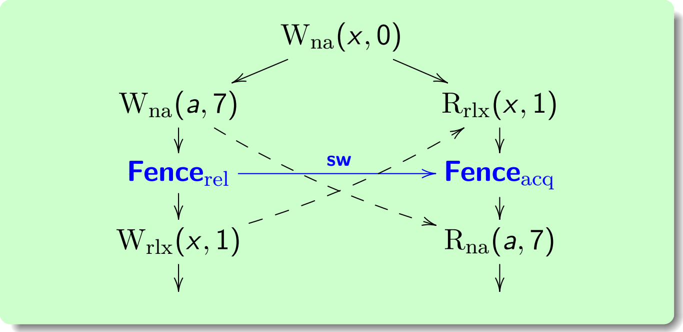
$$\left(\begin{array}{l\|c} a = 7; & \textbf{if } (x.\mathrm{load}(\textit{rlx}) \neq 0)\{ \\ x.\mathrm{store}(1, \textit{rlx}); & \textbf{print}(a);\} \end{array}\right)$$



Message passing with C11 memory fences

$$\textbf{int}\ a;\ \ \textbf{atomic\_int}\ x = 0;$$

$$\left(\begin{array}{l||l} a = 7; & \textbf{if}\ (x.\mathsf{load}(rlx) \neq 0)\{ \\ \color{blue}{\mathsf{fence}(release);} & \quad\quad \color{blue}{\mathsf{fence}(acq);} \\ x.\mathsf{store}(1, rlx); & \quad\quad \textbf{print}(a);\ \} \end{array}\right)$$



Reasoning about fences

Introduce two 'modalities' in the logic.

$$\{P\}\ \mathsf{fence}(\mathit{release})\ \{\triangle P\}$$

$$\{\nabla P\}\ \mathsf{fence}(\mathit{acq})\ \{P\}$$

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ t := x.\mathsf{load}(\mathit{rlx})\ \{\mathbf{R}_{\mathcal{Q}[t:=\mathsf{emp}]}(x) * \nabla \mathcal{Q}(t)\}$$

$$\{\mathbf{W}_{\mathcal{Q}}(x) * \triangle \mathcal{Q}(v)\}\ x.\mathsf{store}(v, \mathit{rlx})\ \{\mathbf{W}_{\mathcal{Q}}(x)\}$$

Reasoning about fences

Let $\mathcal{Q}(v) \stackrel{\text{def}}{=} v = 0 \lor \&a \mapsto 5$.

$$
\begin{array}{c}
\{\&a \mapsto 0 * \mathbf{W}_{\mathcal{Q}}(x) * \mathbf{R}_{\mathcal{Q}}(x)\} \\
\left(\begin{array}{l|l}
\{\&a \mapsto 0 * \mathbf{W}_{\mathcal{Q}}(x)\} & t = x.\mathsf{load}(\mathit{rlx}); \\
a = 5; & \{\nabla(t = 0 \lor \&a \mapsto 5)\} \\
\{\&a \mapsto 5 * \mathbf{W}_{\mathcal{Q}}(x)\} & \mathbf{if}\ (t \neq 0) \\
\mathsf{fence}(\mathit{release}); & \quad \mathsf{fence}(\mathit{acq}); \\
\{\triangle(\&a \mapsto 5) * \mathbf{W}_{\mathcal{Q}}(x)\} & \quad \{\&a \mapsto 5\} \\
x.\mathsf{store}(1, \mathit{rlx}); & \quad \mathbf{print}(a);\ \} \\
\{\mathsf{true}\} & \{\mathsf{true}\}
\end{array}\right)
\end{array}
$$

Why two modalities?

Consider the program, where initially $x = y = 0$:

$$
\begin{array}{l}
a = 5; \\
\text{fence}(\mathit{release}); \\
x.\text{store}(1, \mathit{rlx});
\end{array}
\;\Bigg\|\;
\begin{array}{l}
t = x.\text{load}(\mathit{rlx}); \\
\textbf{if } (t \neq 0) \\
\quad y.\text{store}(1, \mathit{rlx});
\end{array}
\;\Bigg\|\;
\begin{array}{l}
t' = y.\text{load}(\mathit{rlx}); \\
\textbf{if } (t' \neq 0) \; \{ \\
\quad \text{fence}(\mathit{acq}); \\
\quad \textbf{print}(a); \\
\}
\end{array}
$$

If $\nabla P \Rightarrow \triangle P$, we can 'verify' this program.
But the program is racy.

Initially $a = x = 0$.

$a = 5;$	$t = x.load(consume);$
$x.store(release, \&a);$	$\textbf{if}\ (t \neq 0)\ print(*t);$

This program cannot crash nor print 0.

Justification:

$$
\begin{tikzcd}
\mathrm{W}_{\mathrm{na}}(a,5) \arrow[d] \arrow[dr, dashed] & \mathrm{R}_{\mathrm{con}}(x,\&a) \arrow[d] \\
\mathrm{W}_{\mathrm{rel}}(x,\&a) \arrow[ur] & \mathrm{R}_{\mathrm{na}}(a,5)
\end{tikzcd}
$$

Release-consume synchronization

Initially $a = x = 0$. Let $J(t) \stackrel{\text{def}}{=} t = 0 \vee t \mapsto 5$.

$$
\begin{array}{l||l}
\{\&a \mapsto 0 * \mathbf{W}_J(x)\} & \{\mathbf{R}_J(x)\} \\
a = 5; & t = x.\mathit{load}(\mathit{consume}); \\
\{\&a \mapsto 5 * \mathbf{W}_J(x)\} & \{\nabla_t(t = 0 \vee t \mapsto 5)\} \\
x.\mathit{store}(\mathit{release}, \&a); & \textbf{if } (t \neq 0)\ \mathit{print}(*t);
\end{array}
$$

This program cannot crash nor print 0.

Index the ∇ with program variable t.
t data dependence $\implies$ locally open ∇_t.

Proposed rules for consume accesses

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ t := x.\mathrm{load}(\mathit{cons})\ \{\mathbf{R}_{\mathcal{Q}[t:=\mathrm{emp}]}(x) * \nabla_t\, \mathcal{Q}(t)\}$$

$$\frac{\{P\}\ C\ \{Q\} \qquad C \text{ is basic command mentioning } t}{\{\nabla_t\, P\}\ C\ \{\nabla_t\, Q\}}$$

Question: Is the following valid?

$$\{\mathbf{W}_{\mathcal{Q}}(x) * \nabla_t \mathcal{Q}(v)\}\ x.\mathrm{store}(v, \mathit{rel});\ \{\mathbf{W}_{\mathcal{Q}}(x)\}$$

Release-acquire too weak in the presence of consume

Initially $x = y = 0$.

$$
\begin{array}{l||rl}
 & & \textbf{while}\ (x.\mathit{read}(\mathit{consume}) \neq 1); \\
a = 1; & & y.\mathit{store}(1, \mathit{release}); \\
x.\mathit{store}(1, \mathit{release}); & (*) & \textbf{while}\ (y.\mathit{load}(\mathit{acquire}) \neq 1); \\
 & (*) & a = 2;
\end{array}
$$

C11 deems this program racy.

- Only different thread rel-acq synchronize.

> **What goes wrong in PL:**
> On ownership transfers, we must prove that we don't read from the same thread.

Release-acquire too weak in the presence of consume

Initially $x = y = 0$.

$$
\begin{array}{l||rl}
 & & \textbf{while } (x.read(consume) \neq 1); \\
a = 1; & & y.store(1, release); \\
x.store(1, release); & (*) & \textbf{while } (y.load(acquire) \neq 1); \\
 & (*) & a = 2;
\end{array}
$$

C11 deems this program racy. But, it is not racy:

- On x86-TSO, Power, ARM, and Itanium.
- Or if we move the $(*)$ lines to a new thread.

So, drop the “different thread” restriction.

Summary so far

We know how to reason about:

- Release-acquire
- Consume reads
- C11 memory fences

We found a number of bugs in the model:

- Dependency cycles (also in [Batty et al. '03])
- Same thread rel-acq don't synchronize
- Semantics of SC accesses odd and too weak…

  …when mixed with non-SC accesses
- Release sequences too strong

Soundness proof challenges

- Assertions in heaps

  $\Longrightarrow$ Store syntactic assertions (modulo $*$-ACI)

- No (global) notions of state and time

  $\Longrightarrow$ Define a *logical* local notion of state

  $\Longrightarrow$ Annotate hb edges with logical state

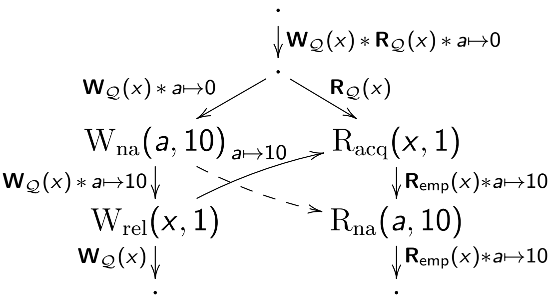
- No operational semantics

  $\Longrightarrow$ Use the axiomatic semantics

  $\Longrightarrow$ Induct over max hb-path distance from top

Basic structure

- Annotate hb edges of executions with heaps.



- Local annot. validity: $\sum \mathit{ins} + \text{node-effect} = \sum \mathit{outs}$.
- Configuration safety: can extend a valid annotation for n further events.

A key lemma

Definition (Pairwise independence)

$\mathcal{T}$ is pairwise independent iff $\forall (a, a'), (b, b') \in \mathcal{T}$, $(a', b) \notin hb^*$.

Lemma (Independent heap compatibility)

If hmap is a valid annotation, and $\mathcal{T} \subseteq hb$ is pairwise independent, then $\bigoplus_{x \in \mathcal{T}} hmap(x)$ is defined.

Conclusion

Formal reasoning about weak memory is possible & not too difficult.

We're not quite there yet; there's still a lot to do:

Liveness, refinement, tool support, . . .

A final remark

Relaxed program logics
are a useful tool for
understanding
weak memory models